The Boost interval arithmetic library

Hervé Brönnimann [a,1] Guillaume Melquiond [b,2] Sylvain Pion [c,3]

[a] *CIS, Polytechnic University, Six Metrotech, Brooklyn, NY 11201, USA.*

[b] *École Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon cedex 07, France.*

[c] *Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany.*

Abstract

We report on the design of the Boost interval arithmetic library, a C++ library designed to efficiently handle mathematical intervals in a generic way. The design of the library is unique in that it uses policies to specify the variable behaviors: rounding, checking, comparisons. As a result, with the proper policies, our interval library is able to emulate almost any of the specialized libraries available for interval arithmetic. This library is openly available at `www.boost.org`. Using this library, we then examine interval-based filters to compute the sign of a determinant, proposed by Burnikel and two authors of the library, and revisit and extend their experiments. We also illustrate other uses of the library.

Key words: Interval arithmetic, library, generic programming, policy-based design, determinant sign, filter.

1 Introduction

Interval computations, known as either *Interval Analysis* or *Interval Arithmetic* (both abbreviated as IA), are a way to extend the usual arithmetic on numbers to intervals on these numbers. The domains of applications of IA are wide and varied. To be useful, however, one usually requires the *inclusion property*: the extension $[f]$ to intervals of a function f is an interval

[1] Work by the first author was supported by NSF CAREER Grant CCR-0133599.
[2] Work by the second author was partially accomplished during a visit to Polytechnic University, with support from ENS Lyon.
[3] Work by the third author was partially supported by NSF CAREER CCR-0133599 while visiting Polytechnic University, and conducted during a postdoc fellowship of NFS/ITR Grant CCR-0082056 at New York University.

$[f]([x_1], \ldots, [x_n])$ which must contain the image by f of every value $(x_1, \ldots, x_n)$ in the function domain. Ensuring this property requires access to the proper rounding modes of the arithmetic operations (most commonly, this can be controlled when using floating point computations that comply with the IEEE 754 Standard [1]),

There are many implementations of interval computations for different purposes (see section 3.5). Since the intended domains of applications sometimes have different semantics, or requirements, these implementations differ in the details. For instance, the meaning of comparisons may be different (section 3.4), or the behavior in exceptional situations may have different requirements (section 3.3). In some contexts like computer graphics, the mathematically correct inclusion property may not even be desirable due to lower speed, and an inclusion property within the roundoff errors may be acceptable. In others, special hardware support can be used to optimize the underlying rounded computations (section 3.2).

We first present the design of the Boost interval library, which was accepted after a thorough review by the members of the Boost community. Its purpose is to provide a single C++ class template, `interval<T>`, and supporting functions, whose behavior can be adapted to the various contexts mentioned above. We settled for a policy-based design [2] and identified three orthogonal behaviors: *rounding*, *checking*, and *comparisons*. All three have several possible choices and we provide a few concrete policies for each. Almost all possible combinations are possible, and we can emulate a wide collection of interval types with a single design. We present this design in section 3.

In a second part, we put the library to use for an experimental study. In previous work [4], we have used interval arithmetic for floating point filters in geometric computing. To illustrate the basic usage of the library, we revisit the experiments with the Boost interval library. The basic primitive there is to compute the sign of a determinant, using either the naïve approach or the *a posteriori* algorithm. In section 4, we recall the algorithms, and provide a short experimental study that extends the results of [4]. In particular, we supply many details that were not provided in their paper, and try a few approaches that were not implemented.

2 Background on interval arithmetic

Interval arithmetic deals with *intervals*: closed and convex sets of a totally ordered field $\mathbb{F}$, called the *base number type*. We denote by $[x] = [\underline{x}, \overline{x}]$ an interval, $\underline{x}$ is the *lower bound*, and $\overline{x}$ the *upper bound*. Basic interval arithmetic is presented in many references (e.g. [7,12,16]). We only review the basics as necessary to understand the scope of the Boost interval library.

Bounds. The base number type can be the real numbers $\mathbb{R}$, but it is not necessary. It can be a subset $\mathbb{F}$ of $\mathbb{R}$, such as the floating-point numbers (e.g., the double precision binary floating point reals of the IEEE Standard 754), rational numbers, even integers. Or it can also be a superset of those (Puiseux series, infinitesimal extensions, etc.).

Operations. Operations and functions are extended to interval functions by the inclusion property: $f([x]) = [f(x)] = \{f(x) \mid x \in [x]\}$. For instance, if both $[x] = [\underline{x}, \overline{x}]$, $[y] = [\underline{y}, \overline{y}]$ are bounded intervals, we set

$$
\begin{aligned}
[x] + [y] &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\
[x] - [y] &= [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \\
[x] \times [y] &= [\min\{\underline{xy}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{xy}\}, \max\{\underline{xy}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{xy}\}] \\
[x] \ / \ [y] &= [x] \cdot [1/\overline{y}, 1/\underline{y}] \text{ if } 0 \notin [y].
\end{aligned}
$$

Unbounded intervals will be denoted by using an infinite bound in the correct place ($[x, +\infty]$ is $\{y \in \mathbb{F} \mid y \geq x\}$, $[-\infty, x]$ is $\{y \in \mathbb{F} \mid y \leq x\}$ and $[-\infty, +\infty]$ is $\mathbb{F}$). There is no projective infinity [16] with this representation. Finally, there is an empty interval $\emptyset$.[4] An operation involving an empty interval will return an empty interval.

For an unbounded interval, the addition and subtraction are trivially adapted thanks to the operations defined on the extended base number type $\overline{\mathbb{F}} = \mathbb{F} \cup \{-\infty, +\infty\}$. For the multiplication, it requires a bit of work. Such an interval arithmetic system is common and a detailed justification of it can be found in [7] for example.

As for interval division, we have defined it by canonical set extension of the division in $\mathbb{F}$. However, the answer may not be an interval when dividing by an interval containing 0. For example, $1/[-1, 1] = [-\infty, -1] \cup [1, +\infty]$. So, two versions can be defined: one of them will answer a single enclosing interval ($[-\infty, +\infty]$ in this example) and the other will answer a union of intervals (at most 2).

[4] The representation of the empty set by a pair of bounds is implementation-dependent and left unspecified. It will usually be represented by a pair $[a, b]$ such that $a \leq b$ is false.

Rounding and reliable computations. In computer arithmetic, we usually work with a subset of the field $\mathbb{F}$, called the *representable* numbers, and this subset may not be a field (e.g. the binary floating point reals of the IEEE Standard 754). Since we cannot always represent exactly the result of an operation on representable numbers, we must therefore compute in each arithmetic step the smallest interval $\Diamond[x] = [\triangledown\underline{x}, \triangle\overline{x}]$ that encloses $[x]$ such that $\triangledown\underline{x}$ and $\triangle\overline{x}$ are representable, an operation called *rounding*. This means that $\triangledown\underline{x}$ (respectively $\triangle\overline{x}$) is the next representable number to $\underline{x}$ (respectively $\overline{x}$) when rounding downwards (respectively upwards). If there is no such representable number, the corresponding infinity $-\infty$ or $+\infty$ should be given to ensure the inclusion property of interval arithmetic. [5]

However, first computing bounds in $\mathbb{F}$ and then rounding them to representable bounds is only an abstract construct. In an actual implementation, rounded operations must be defined such that we can compute $[x] \oplus [y] = [\underline{x{+}y}, \overline{x{+}y}]$ for example. [6]

3 The design of the Boost interval library

The general principle of interval arithmetic described in the previous paragraph is applied to get a C++ class template `interval<T>`. There are many details to fill in, however, such as the definition of an empty interval, what happens for exceptional values of the base types (such floating point NaNs—*not a number*), how to perform the rounding (or not, if so desired). Typically these choices are performed with a particular domain of application in mind.

The goal of the Boost interval library is to provide a *generic* implementation of IA, in that those choices can be specified by the user of the library. For this, we use a mechanism called *policies* [2,15].

3.1 Overview of the policy-based design

The class template `interval<T>` actually has two template parameters, `T` and `Policies`, the second being chosen by default. The second parameter does not influence the data representation of an interval; it is only here to describe the way the various algorithms will handle the data. It contains a reference to two types, one being the rounding policy, the other the checking policy. A

[5] If the implementation does not know anything about infinities, a suitable behavior like throwing an exception should be adopted.
[6] Note that IEEE Standard 754 guarantees that $\underline{x{+}y}$ is the same as $\triangledown(\underline{x} + \underline{y})$, but it suffices in general that $\underline{x{+}y} \leq \triangledown(\underline{x} + \underline{y})$ to ensure the inclusion property.

policy is a class that specifies the behavior. For instance, the rounding policy will have operations for performing all the basic operations on the base type, with a specified rounding mode.[7]

Note that the complete C++ interval type not only contains `T`, but also the policies. This is useful to prevent automatic casts from `interval<T,P1>` to `interval<T,P2>`, e.g., in case the checking policy of `P1` allows empty intervals but not `P2`.

Strictly speaking, comparisons are not a policy, since doing so would have incorporated the comparison into the interval type. We view comparisons not as intrinsically part of the interval type, as it is perfectly legal (and sometimes useful) to compare the same intervals in different ways (for instance, certain comparisons for non-overlapping intervals, then some special treatment for overlapping ones).

3.2 The rounding policy

Since interval operations can be expressed in term of rounded operations on the bounds, the library functions rely on a kernel of arithmetic functions to compute their results. The kernel to use is indicated by the rounding policy.

The library could have directly used a kernel predetermined for each base type. But it is not interesting to only have one kernel for a given base type, nor is it possible to predetermine a kernel for every user-defined base type. For example, suppose the class is an interval with rational bounds and the user wants to compute $[\sqrt{2}]$. No rational bounds can express this interval, so the kernel will choose approximations of these bounds. Choosing among different policies, the user may prefer a rounding policy that computes small rationals for $\sqrt{2}$, or one that gives a better approximation but takes longer, etc.

These kernels are also responsible of all the optimizations specific to the type of bounds; in particular, when manipulating hardware floating-point numbers. Here is a common example of such an optimization. Processor floating-point units are usually unable to efficiently handle rounding mode changes: the latter breaks the execution flow and increases latency. In order to avoid these switches, a sequence of operations can be done with one fixed rounding mode. For example, if the current rounding mode is towards $+\infty$, the sum $a \underline{+} b$ can be computed as $-((-a) \overline{+} (-b))$ without changing the rounding mode. All this can be encapsulated within the arithmetic kernel. The functions on intervals

[7] For example, $\underline{+}$ and $\overline{+}$ are such operations and the library functions rely on this policy to compute them.

do not need to know about these optimizations; they only ask the rounding policy to perform $a \underline{+} b$, no matter how.

Thanks to these two stages, the functions of the library are totally generic and can handle any type of intervals, yet the library is as fast as specialized libraries since the various kernels can be fully optimized for a particular base number type.

3.3 The checking policy

This policy allows the user to choose how the interval class will deal with exceptional cases. In particular, empty intervals may play an important role in some algorithms. However, there are also situations where they cannot occur, or where the user does not want them to be generated (a division by $[0, 0]$ should generate an empty interval but it can also mean there is a bug in the algorithm).

A function deals with empty intervals by asking the checking policy to verify if the input intervals are empty or not. If the user is sure no empty intervals will ever be created, the policy can disable these tests and the compiler will do a lot of dead code elimination.

It is generally safe to think that if no empty interval can be passed as input, no empty interval will be produced by a function, hence the checking can be done away with. But it can also happen that a function needs to output an empty interval (generally when the interval is outside the domain of the function, $\arccos[25, 32]$ for example). In this case, the function will ask the checking policy to create such an interval. Depending on the policy, an empty interval will be created, or an exception will be thrown (since computing $\arccos[25, 32]$ was probably a bug).

Empty intervals are not the only special case, the library functions should also be cautious and test each input for invalid data (a *Not a Number* of the IEEE 754 standard for example). But it is an overhead that is not always desired and that can be turned off by choosing the appropriate policy. For all these exceptional cases, the checking policy allows the user to select a particular behavior of the library.

3.4 Comparisons

When comparing two intervals $[a, b] < [c, d]$, three cases can occur. First, the inequality may be satisfied for any pair of numbers in the intervals: $\forall x \in$

$[a,b]\ \forall y \in [c,d]\ x < y$. If not, it may be satisfied for some pair: $\exists x \in [a,b]\ \exists y \in [c,d]\ x < y$. Finally, it may be satisfied for none. Because of these three cases, it is not possible for a comparison to directly map to a boolean type, the user needs to select a particular mapping.

The comparison scheme is made available through namespace resolution. The main advantage is that the policy is selected locally, so that several incompatible policies may be used in different portions of a same program. The advantage of namespace resolution is that it allows to use the infix notation (operators `<`, `<=`, `>`, `>=`, `==`, `!=`) rather than calling some functions.

Thanks to this model, many comparison semantics can be defined. In particular, the library provides a lexicographic order (no real mathematic meaning but sometimes useful when manipulating datas), an order based on the set inclusion partial order (subset, superset, proper subset, etc). There is also a namespace of operators that do not return a boolean but a tristate value to reflect the previous mapping.

The default behavior of the operators is meant to reflect the results of the operators on the base numbers. So when the comparison is certain, the answer is `true`. When no pair of elements can satisfy the comparison, the answer is `false`. And when no definite answer can be given (generally because the intervals overlap), an exception is thrown.

3.5 Comparison with other C++ libraries

Many libraries and applications provide interval arithmetic. We compare with five of them that are typical implementations. Others can be found on the Interval web page [8].

Profil/BIAS [10] and CGAL [6] only handle bounds of type `double` (double precision floating-point number as defined in the C++ standard). Filib [11] and Sun [14] are a bit better in the sense that they are able to handle the three floating-point types `float`, `double`, and `long double` thanks to a template parameter. So these four libraries are restricted to hardware floating-point numbers.

MPFI [13] handles multi-precision floating-point numbers (as given by the MPFR library). So there is a bit of flexibility since the precision can be chosen. However, one more time, only floating-point numbers are supported and none of these libraries are able to manipulate intervals with rational or integer bounds for example.

A library like Profil/BIAS also suffers from always switching rounding mode.

Thanks to BIAS level 1 and 2 techniques, the number of switch is reduced when computing with matrices and vectors. But not all programs manipulate matrices and they will then suffer from performance degradation. CGAL, Sun and Filib provide better rounding mode handling to avoid this overhead.

Speed is not the only concern, validity of the results can also be a problem. For example, not all these libraries are able to correctly compute the product $[-1, 0] \times [5, +\infty]$ and the result will be absurd[8] (Profil/Bias will return $[-\infty, \mathrm{NaN}]$ for example, although the correct answer would have been $[-\infty, 0]$).

These libraries also have specific behavior in regard to empty intervals. They may test for them and handle them (MPFI) or throw an exception. They can also ignore them and produce invalid results. The behavior is fixed and sometimes not clearly defined; and moreover, the user cannot change it. Another frustrating detail, none of these libraries allows to use specially crafted infix comparison operators and the user can only rely on functions.

By using the correct policies, the user should be able to emulate all the existing libraries with this library, without sacrificing the precision, validity and speed of the computations. But, more important, the user is able to define a lot of new behaviors that were impossible to accomplish with these libraries.

3.6 Example of use of the library

Given a polynomial P (represented by an array `P` and its size `sz`) and a value x, the following example computes the sign of $P(x)$. The answer is 1 if $P(x)$ is positive, -1 if $P(x)$ is negative, and 0 if the function does not compute the answer safely.

```
int sign_polynomial(double x, double P[], int sz) {
  // definition of an interval
  typedef interval<double> I_aux;

  // rounding optimization
  I_aux::traits_type::rounding rnd;
  typedef unprotect<I_aux>::type I;

  // Horner's scheme
  I y = P[sz - 1];
  for(int i = sz - 2; i >= 0; i--)
    y = y * x + P[i];
```

[8] Having an interval with an infinite bound can easily result from an overflow when dealing with floating-point numbers and an interval arithmetic library should be able to handle such situations.

```
    // sign evaluation
    using namespace compare::certain;
    if (y > 0.) return 1;
    if (y < 0.) return -1;
    return 0;
  }
```

There are four steps in this function. First, define the type `I_aux`, an interval type whose bounds are double precision floating-point numbers. Second, indicate to the library to use an optimization for the whole function as explained below. Next, compute the value y of $P(x)$ by Horner's scheme. The only difference with the non-interval case is that y is not of type `double` but of type `I`; other than that, the program is identical. And finally, choose the certain comparison scheme and return the sign.

As shown by this example, the code is short and easy. It could even be shorter if the optimization was not used (to remove the optimization, just delete the two corresponding lines and use only one interval type for the whole function). This optimization is used to inform the library that past this point (more precisely during the life of the object `rnd`) the user will only use mathematical functions involving IA and that the library is allowed to use whatever possible mean to speed up the computations. It requires a new interval type to be defined since this information will be passed thanks to the interval policies.

If the processor supports hardware floating-point computations, the optimization will consist in setting the rounding mode upwards for the whole life of `rnd` and doing all the computations with this mode. Note that the destruction of `rnd` upon return will reset the rounding mode to what it was before.

4 Sign of a determinant

4.1 Statement of the problem and algorithms

We now apply IA to a well-studied problem in linear algebra. Much of the work in this section is a recall from [4].

Statement. Given an $n \times n$ matrix A, we would like to safely compute the sign of its determinant $\det(A)$, or conclude that we cannot do so safely with the available precision. Such an evaluation is called a filter and is heavily relied on to speed up geometric computations [6]. Interval arithmetic is very well suited to that [4], and we recall two algorithms developed in that article for that purpose.

Reference algorithm. The reference algorithm will compute a decomposition $P \cdot A = L \cdot U$ where P is a permutation matrix, L is lower triangular with only 1 on the diagonal and U is upper triangular. The strategy is standard Gaussian elimination, with a partial pivoting choosing the biggest element in absolute value. The determinant of P is ± 1 and can be computed without error since P is only a permutation matrix built during the partial pivoting. Since L is triangular and all its diagonal elements are 1, $\det(L)$ is 1 and does not have to be computed (and so it is not necessary to compute L). Finally, since U is also triangular, $\det(U) = \prod_i u_{i,i}$. Thus the reference algorithm answers $\det(P) \times \prod_i \operatorname{sign}(u_{i,i})$. The answer may or may not be correct depending on how well-conditioned A is w.r.t. the base precision available.

As is well known, the whole algorithm requires $n^3/3 + O(n^2)$ operations (1 operation = 1 addition + 1 multiplication), and can be carried in place if A can be destroyed by the algorithm (otherwise a copy of A needs to be made).

A naïve interval extension. The previous algorithm can be performed with intervals, computing a decomposition $P \cdot A = [L] \cdot [U]$. With intervals, there is an additional constraint: the pivot should not contain zero. The reason is trivial: since the pivot will become one of the diagonal elements of U, if it contains zero, the sign of the determinant will be unknown. If no suitable pivot can be found, the algorithm returns early that it cannot compute the sign safely. Otherwise it computes an enclosure of the determinant which does not contain 0, and returns $\det(P) \times \prod_i(\operatorname{sign}[u_{i,i}])$.

The temporal complexity is now $n^3/3$ interval additions and multiplications. As observed in [4], interval operations incur a cost overhead of about 2.6 to 3 times slower than the reference algorithm, if the base type is a builtin floating point. Also, the algorithm needs to store the matrix $[A]$ and its interval elements, so it requires twice the space necessary for the reference version. Hence, this algorithm will hit the cache limit of the processor sooner, and an additional overhead can be expected for the temporal complexity in some cases.

An *a posteriori* algorithm. The previous algorithm may be too cautious, in that the answer of the reference algorithm may be correct (but the previous algorithm cannot certify it). In order to obtain a better success rate, we apply an idea of [4], to verify that the sign returned by the reference algorithm is correct. This certification can be done *a posteriori* by instructing the reference algorithm to also compute (numerically without intervals) approximations L_{inv} and U_{inv} of L^{-1} and U^{-1}.

Lemma 1 [4] *Let F be a square $n \times n$ matrix, I the identity matrix dimension n, and $||\cdot||$ a matrix norm (i.e., a norm such that $||Ax|| \leq ||A|| \cdot ||x||$). If $||F - I|| < 1$, then* $\det(F) > 0$.

To apply the lemma, note that $B = U_{\text{inv}} L_{\text{inv}} P$ should be equal to A^{-1}, and thus their determinants should have the same sign. But the sign of $\det(B)$ is precisely the same as answered by the reference algorithm. In general, $\det(A) = \det(B)^{-1}\det(BA)$ implies that $\det(A)$ has same sign as $\det(B)$ if $\det(BA) > 0$. This can be ensured by checking that the norm of $AB - I$ is less than 1, according to the lemma. Of course, to be able to guarantee the result, one has to use interval arithmetic in computing the norm of $AB - I$.

In conclusion, to ensure that the sign of $\det(A)$ is indeed given correctly by the reference algorithm, it suffices to check that the matrix $[U_{\text{inv}} L_{\text{inv}} PA - I]$, with all matrix operations evaluated with intervals, has a norm less than 1.

The temporal complexity now becomes that of the reference algorithm, modified for computing the inverse, plus the IA for the norm. As noted in [4], this takes $n^3 + O(n^2)$ operations in $\mathbb{F}$ plus $n^3 + O(n^2)$ interval operations, because the computation of $U_{\text{inv}}(L_{\text{inv}}(PA))$ involves two products of a full matrix by a triangular matrix.

The spatial complexity can be improved from the algorithm of [4], by choosing the $||\cdot||_\infty$ or $||\cdot||_1$ for the *a posteriori* step. Indeed, computing $||[((PA)U_{\text{inv}})L_{\text{inv}} - I]||_\infty$ can be done one row after another, so only additional space for one row is required. Or equivalently, one may compute $||[U_{\text{inv}}(L_{\text{inv}}(PA)) - I]||_1$ one column after another.

By choosing one of these two formulas, the temporal complexity does not change and the spatial complexity is now of the same order than the complexity for the naïve algorithm; the only difference in space is the area necessary for the additional row or column. The choice between the two formulas should then be made according to the data layout (row-major or column-major) in order to maximize the spatial and temporal locality of the memory accesses.

4.2 Engineering IA and the algorithms

In theory, IA computations take at least twice as long as the base type; indeed, the operations have to be carried out for both bounds of the intervals. In practice, due to the higher complexity of multiplications and divisions, the overhead is somewhere between 3 and 4, and more if rounding modes and other operations have to be carried out. There exist many solutions for computing the product $[x] \times [y]$. The one chosen in the library consists in classifying the two intervals $[x]$ and $[y]$ in four categories depending of their "signs" and then to do two numbers multiplication to get the resulting interval. Consequently, before the floating-point operations can effectively be done, four conditional instructions need to be executed to get the correct classes of the two input intervals.

What we now show, is that with some knowledge of the context and by carefully organizing the IA computations and selecting the right interval type, it is possible to approach the theoretical overhead of two of IA. All the experiments have been made on a PowerPC 7450 with GNU C++ version 3.2.

Generally speaking, when carrying interval operations in sequence, it is best to use the negation trick mentioned in Section 3.2, to avoid changing the rounding mode. We do this in the default rounding policy for the builtin floating-point types.

The naïve extension to IA of the reference algorithm does not require any engineering, and takes exactly the same number of operations (but this time, interval operations). So the slowdown depends on the speed of IA. Because the first algorithm uses a blend of additions with multiplications, its speed is about 2.6 to 3 times slower than the reference implementation [4].

For the *a posteriori* algorithm, we can improve the bound of [4], by remarking that in those two matrix multiplications, half of the interval multiplications do not require any comparison at all (the two intervals are a single point) or only half the comparisons (one of the intervals is a single point), hence the overhead of IA is only an optimal factor of 2 in those operations. This is where the Boost library offers some advantage compared to the implementation of [4], since it provides mixed-type multiplications. For the other operations, we measure experimentally, as in [4], that the overhead of IA is about 3. Overall, then, this algorithm performs asymptotically at least 9 times more base type operations than the reference algorithm, and we find experimentally that its running time is only a factor 8–10 times slower. Hence, the library techniques did help in achieving an almost optimal theoretical overhead in that case.

For small dimensions, we note that neither algorithm is very good, because pivoting involves tests and is a slow operation. Let's consider 4×4 determinants as a representative of these small determinants. The dimension is small enough for the naïve algorithm to perform far better than *a posteriori*: twice faster for the same precision. So only this algorithm will be considered.

Since the dimension is small, the direct method (developing recursively along a row or a column) is feasible. With this method, the 2×2 sub-determinants are precomputed in order to avoid computing them multiple times (two times each in fact) during the recursion.

It is also possible to evaluate the determinant thanks to a 2×2 block decomposition. Indeed, if we note A, B, C and D the four 2×2 blocks, $|A|$ the determinant of A and A' the cofactor matrix of A, then we get:

$$\det \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \det \begin{pmatrix} A & B \\ 0 & D - C \cdot A^{-1} \cdot B \end{pmatrix} = \frac{1}{|A|} \det \left(|A| D - C \cdot A' \cdot B \right)$$

As for the number of interval operations, the direct method takes 28 interval additions and 17 interval multiplications, vs. 24 and 14 for the block decomposition, and 14 additions+multiplications and 6 divisions on intervals for the *a posteriori* method (not counting comparisons for the search for the best pivot). The experimentations showed that the fastest method is the block decomposition. The direct method is a bit slower. And the LU-decomposition is 2 to 3 times slower. But the benefits of blocked determinants also include better precision (next section).

4.3 Experimental results

We replicate some experiments of [4] and develop others to demonstrate the effectiveness of the Boost library approach.

For the evaluation of the naïve and *a posteriori* algorithms, our results are similar to those of [4], but a bit better thanks to the better implementation of the library. For instance, consider the singular matrix which all elements are 1 and perturb this matrix so that it becomes regular. The perturbation on each element is bounded by 2^{-p}. Table 1 shows the value of p for which a filter fails to return a result for at least 50% of the matrices of a random population. As observed in [4], the number of bits lost during the computations with the first algorithm is proportional to the dimension of the matrix. As for the second filter, the loss is not linear since interval arithmetic is restricted to the product of three matrices. So this test shows that the second filter is more adapted to big matrices than the first.

Dimension n	4	6	8	10	12	14	16	24	32	40	48	56
Naïve	50	47	45	43	41	39	37	29	22	14	7	-
a posteriori	49	47	46	46	45	45	44	43	42	41	41	40

Table 1
The minimal value of p for which the filters fail for at least 50% of the cases.

For the case of 4×4 determinants, the *a posteriori* almost reaches the maximal precision ($p = 49$). The direct method becomes inefficient as soon as $p = 17$; and the block method reaches $p = 25$. So the block method is a reasonable compromise, preferable to the row expansion for both its precision and speed, and to the *a posteriori* when speed matters.

In addition, we perform experiments with the classical Hilbert matrix [5]

$$A_n = \left(\frac{1}{i+j+1}\right)_{0 \le i,j < n}.$$

Its determinant is much smaller than any of its entries, but still positive, and it is well known as an example of an ill-conditioned matrix. With double precision floating-point numbers, the naïve algorithm fails as soon as $n = 10$ and the *a posteriori* fails when the dimension reaches $n = 13$. So the maximal dimension can be considered to be really low; and it is not really exceptional: whatever happens, if a floating-point algorithm fails and gives a bad result, no interval method can guarantee it.

4.4 Extension: Dealing with imprecise input

The previous filters suppose the input is a matrix of base type numbers. If the input is not representable, it does not make sense to round it to the closest representable matrix, as the algorithms will only guarantee the sign of the determinant of that matrix, and not of the input. Hence, the input has to be an interval matrix. Another situation that requires an interval input is when the input cannot be determined with sufficient precision.

The first algorithm uses intervals to do the whole computation of the sign. Moreover, its first step is simply to convert A to $[A]$. So this algorithm can directly be used if an interval enclosure of A is known rather than A itself. This method does not incur any overhead.

The situation is more complicated for the second algorithm. Indeed, the whole point of the filter is to use base number operations to do the initial steps. If only an enclosure $[A]$ is known, a representative A' of $[A]$ is enough to run these initial steps of the algorithm. It is then possible to certify the sign of $\det[A]$ by computing the norm (with intervals) of $U_{\text{inv}} L_{\text{inv}} P[A] - I$ and checking that it is less than 1. This time, there may be a slight overhead in choosing A' (the median of A for example) but it does not change the overall complexity of the algorithm: there is still $n^3 + O(n^2)$ operations in $\mathbb{F}$ and $n^3 + O(n^2)$ interval operations.

In some variants, both A and an absolute error E on the coefficients of A are known. By writing $[A] = A + [-E, E] \cdot I$, it suffices at the end of the algorithm to compare the norm of $[U_{\text{inv}} L_{\text{inv}} PA - I]$ not with 1, but with $1 - ||U_{\text{inv}} L_{\text{inv}}|| \cdot E$. Unfortunately, this method requires the algorithm to compute $[U_{\text{inv}} L_{\text{inv}}]$ to obtain the norm (which can be reused to compute $[U_{\text{inv}} L_{\text{inv}} PA]$). Since the matrix computations are done in a different order, the total number of interval operations becomes $\frac{4}{3}n^3 + O(n^2)$.[9] For large values of n, it might be preferable to precompute $[A] = A + [-E, E] \cdot I$ and apply the former extension to $[A]$, using A as the matrix A'.

5 Conclusion

This paper introduces a new interval arithmetic library, and illustrates its use in a study of some efficient methods for computing signs of determinants. It is a generic C++ library able to handle any kind of bounds: hardware floating-point numbers, rationals, multi-precision software number, etc., and emulate a wide variety of behaviors (thus encapsulating various existing libraries). Performance has not been sacrificed to this genericity, however. In particular, with hardware floating-point numbers, on a complex algorithm (the second filter of section 4 for example), the overhead of interval arithmetic can be made optimal, even when the algorithm involve intensive computations like matrix operations.

Having a good interval arithmetic library, it becomes possible to engineer algorithms carefully without trading precision for speed. Other techniques may then be implemented with benefits. For determinant sign computation, the use of *a posteriori* as recommended by [4] improves on the precision of the intervals used. The method only uses floating-point arithmetic and hopes the computations are stable enough to provide an accurate results (thanks to error compensation). Interval arithmetic is only used to validate the result. So, whatever the inherent complexity of the computations, if the validation steps are simple enough, the intervals will not grow too much. The direct method (developing recursively along the last row or last column) only works for very small determinants ($n \leq 3$). The 2×2 block decomposition is faster and more precise for $n = 4$, but lacks precision compared with the interval LU-decomposition and, depending on the kind of input, the direct algorithm may be a better choice. Finally, for bigger determinants, the *a posteriori* method gives the most precise results.

[9] Indeed, first multiplying U_{inv} and L_{inv} and then their product by PA requires around $(\frac{1}{3} + 1)n^3$ interval multiplications. On the other hand, successively multiplying PA by the two triangular matrices only requires around $(\frac{1}{2} + \frac{1}{2})n^3$ multiplications. Moreover, in this version, half of the interval multiplications do not require any comparison at all (the intervals are single points); when the triangular matrices are first multiplied, only a quarter does not require any comparison at all.

References

[1] ANSI/IEEE. *IEEE Standard 754 for Binary Floating-Point Arithmetic*. IEEE, New York, 1985

[2] A. Alexandrescu. *Modern C++ Design.* Addison Wesley, 2001.

[3] Boost. *The Boost C++ Libraries.* `www.boost.org`

[4] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Disc. Appl. Maths* 109:25–47, 2001.

[5] M.-D. Choi. Tricks or treats with the Hilbert matrix. *Amer. Math. Monthly*, 90:301-312, 1983.

[6] CGAL. *Computational Geometry Algorithms Library.* `www.cgal.org`

[7] T. Hickey and Q. Ju and M. H. Van Emden, Interval arithmetic: From principles to implementation. *J. ACM*, 48(4):1038–1068, 2001.

[8] *Interval Computations.* `http://www.cs.utep.edu/interval-comp/main.html`

[9] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis, with Examples in Parameter and State Estimation, Robust Control and Robotics.* Springer-Verlag, 2001.

[10] O. Knueppel. PROFIL/BIAS - A Fast Interval Library. *COMPUTING* Vol. 53, No. 3-4, p. 277-287. `http://www.ti3.tu-harburg.de/knueppel/profil/`

[11] M. Lerch, G. Tischler, J. Wolff von Gudenberg, W. Hofschuster, and W. Krämer. *FILIB++ Interval Library.* `http://www.math.uni-wuppertal.de/org/WRST/software/filib.html`

[12] R.E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1966.

[13] N. Revol and F. Rouillier. *MPFI 1.0, Multiple Precision Floating-Point Interval Library.* `http://www.ens-lyon.fr/~nrevol/mpfi_toc.html`

[14] Sun Microsystems. *C++ Interval Arithmetic Programming Reference.* `http://docs.sun.com/db/doc/806-7998`

[15] D. Vandevoorde and N. Josuttis. *C++ templates: the Complete Guide.* Addison Wesley, Boston, 2002.

[16] G. William Walster. *The Extended Real Interval System.* Manuscript, 1998. Preprint available at `http://www.mscs.mu.edu/~globsol/walster-papers.html`.