Doc No: N1843=05-0103

A Proposal to add Interval Arithmetic to the C++ Standard Library

Hervé Brönnimann[*] Guillaume Melquiond[†] Sylvain Pion[‡]

2005-08-01

Contents

Contents **1**

I Motivation and Scope **2**

II Impact on the Standard **2**

III Design Decisions **3**

IV Proposed Text for the Standard **7**
26.6 Interval numbers . . . 7
26.6.1 Header `<interval>` synopsis . . . 8
26.6.2 `interval` class template . . . 9
26.6.3 `interval` numeric specializations . . . 10
26.6.4 `interval` member functions . . . 12
26.6.5 `interval` member operators . . . 13
26.6.6 `interval` non-member operations . . . 14
26.6.7 `interval` comparisons . . . 15
26.6.8 `interval` IO operations . . . 16
26.6.9 `interval` value operations . . . 16
26.6.10 `interval` algebraic operations . . . 17
26.6.11 `interval` set operations . . . 17
26.6.12 `interval` static value operations . . . 18
26.6.13 `interval<bool>` specialization . . . 18
26.6.14 `interval<bool>` member functions . . . 19
26.6.15 `interval<bool>` conversion to `bool` . . . 20
26.6.16 `interval<bool>` non-member operations . . . 20
26.6.17 `interval<bool>` static value operations . . . 21

V Possible extensions **21**
V.1 `<cmath>` functions . . . 21

VI Examples of usage of the `interval` class. **22**
VI.1 Unidimensional solver . . . 22
VI.2 Multi-dimensional solver . . . 23

VII Acknowledgements **25**

References **25**

[*]CIS, Polytechnic University, Six Metrotech, Brooklyn, NY 11201, USA. `hbr@poly.edu`
[†]École Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon cedex 07, France. `guillaume.melquiond@ens-lyon.fr`
[‡]INRIA, BP 93, 06902 Sophia Antipolis cedex, France. `Sylvain.Pion@sophia.inria.fr`

I Motivation and Scope

Why is this important? What kinds of problems does it address, and what kinds of programmers is it intended to support? Is it based on existing practice? Is there a reference implementation?

Interval arithmetic (IA) is a basic tool for certified mathematical computations. Basic interval arithmetic is presented in many references (e.g. [5, 11, 15]). Rather than recalling the mathematical definition of IA, we refer the reader to our accompanying paper in which we describe the design of the Boost.Interval library. That paper [2] contains a definition of IA in the mathematical context of an ordered field (not necessarily the reals, although this proposal only touches the basic floating point types), along with a discussion of the interval representations, rounding modes, basic operations on intervals, including divisions, unbounded and empty intervals, and possible comparisons schemes.

Concerning previous work, there exist many implementations of IA (see [8, 4, 9, 13, 14, 12] for six typical C++ implementations; more can be found on the Interval web page [6], including for other languages). In particular, the ancestor of this proposal is the Boost interval library [3]. They provide similar but mutually incompatible interfaces, hence the desire to define a standard interface for this functionality.

There are several kinds of usage of interval arithmetic [6, 7, 10]. There is a web page gathering information about interval computations [6]. Among other things, it provides a survey on the subject and its application domains[1]. We can list a few here, while noting that this illustrative list is in no way exhaustive[2]:

— Controling rounding errors of floating point computations at run time.

— Solving [systems of] [plain, linear, or differential] equations using interval analysis.

— Global optimization (e.g., finding optimal solutions of multi-dimensional not-necessarily-convex problems).

— Certified mathematical proofs (e.g., Hales' recent celebrated proof of Kepler's conjecture[3]).

IA can be implemented in a library, and usually requires rounding mode changes functions, which are available in `<cfenv>`. Having compiler support can greatly speed up the implementation (by eliminating redundant rounding mode changes for example).

Why standardize it?

— The functionality is needed in many areas;

— There are many existing implementations, all with different design choices;

— A basic version is not hard to implement and can be done with only standard components (no need to have auxiliary libraries unless the `<cmath>` extensions are also standardized); and

— Standardization provides an opportunity to have better and more optimized implementations.

A prototype implementation of this proposal and some example programs can be found at `http://www-sop.inria.fr/geometrica/team/Sylvain.Pion/cxx/`.

II Impact on the Standard

What does it depend on, and what depends on it? Is it a pure extension, or does it require changes to standard components? Can it be implemented using today's compilers, or does it require language features that will only be available as part of C++0x?

[1] `ftp://interval.louisiana.edu/pub/interval_math/papers/papers-of-Kearfott/Euromath_bulletin_survey_article/survey.ps`

[2] Visit `http://www.cs.utep.edu/interval-comp/appl.html` for more examples with links

[3] `http://www.math.pitt.edu/~thales/kepler98/`

It is a pure extension to the standard library.

III Design Decisions

Why did you choose the specific design that you did? What alternatives did you consider, and what are the tradeoffs? What are the consequences of your choice, for users and implementers? What decisions are left up to implementers? If there are any similar libraries in use, how do their design decisions compare to yours?

Design overview:

The basic design aims at introducing a single class template `interval<T>` which guarantees the inclusion property. Like `std::complex<T>`, we decided to support the three built-in floating point types and leave the rest unspecified. We decided to support empty intervals, because they can be integrated easily into the proposal. We decided to support equality and relational comparisons that extend the comparisons on the base type `T`. This implies dealing with comparisons of empty or overlapping intervals. Both can be made to work very naturally by also providing an `interval<bool>` as the result of such comparisons, and exceptions can be avoided up to this level. In order to use comparisons in conditional statements, a conversion from `interval<bool>` to `bool` is provided, and only at that level is an exception thrown, when it involves an empty or indeterminate boolean interval. The behavior on out-of-range argument values (in `sqrt`, for instance) is a silent and no-exception behavior, which returns the empty interval.

Alternatives and trade-offs:

Not supporting empty intervals implies all kinds of decisions about out-of-range argument values (in the mathematical functions, but also including the constructors and intersection functions). Mostly, it raises the question of exception-throwing, which we have tried to avoid whenever possible. It turns out empty intervals solve all these problems and still return enough information to allow the other approaches by manual testing.

For comparisons, there are several well-known semantics that return a `bool`: certainly and possibly comparisons for the extensions of the order on the base type, or the completely different set inclusion semantics. The problem is what to do in the case of overlapping intervals, and exception seems about the only alternative for the extensions, which is why some systems prefer the set inclusion semantics, which do not raise exceptions. We already provide support for the set inclusion semantics in the form of function templates, and the use of `interval<bool>` solves the problem of exceptions and allows for both certainly or possibly comparisons by hand.

Decisions left to implementers:

The representation of the empty interval is the most obvious one, although the I/O representation is fixed in this proposal. (See the item below for a discussion of possible choices.) The width and midpoint of an empty interval are also implementation-defined (we recommend a NaN on systems that support it). The value of `whole()` requires that `T` has an infinity (in `std::numeric_limits<T>`).

Comparison with existing libraries:

Rather than just putting together the common subset of features provided by existing C++ interval arithmetic libraries, we tried to propose a consistent yet complete superset of features. Please note that the Boost.Interval library provides these features (such as whether to support empty intervals or not, how to deal with rounding, the meaning of comparisons, user-defined types, etc.) via a policy-based design. This proposal is aimed at reducing the accompanying complexity (for the implementer, but also the user), by making reasonable and conservative choices.

In addition to the Boost.Interval library that acted as a sandbox for this proposal, we considered the C++ libraries PROFIL, filib++, Gaol, and Sun interval library. Except for Boost and filib++, none provides support for user-defined types; neither does this proposal. PROFIL and Gaol only provide support for double-precision intervals; we provide all three built-in floating point types.

There are two usual ways of handling hardware rounding: either you handle it transparently in each interval operation, or you set it globally and require the user to be very careful with his own floating-point computations. In Boost and filib++, either of these two behaviors can be selected through template parameters. In PROFIL, one of them is selected through a macro definition. In Gaol, only global rounding is available. Finally, in Sun and in this proposal, rounding mode switches are handled transparently, they never leak outside of interval operations.

Except for PROFIL, all these implementations correctly support infinite bounds. They also handle empty input intervals the same way this proposal does: the result of an operation involving an empty interval is an empty interval. With respect to division by an interval containing 0, the libraries Boost and Gaol and this proposal provide the tighter intervals (zero or semi-infinite intervals whenever possible).

There is no comparison order on intervals that matches the natural total order defined on the base type. Yet a lot of equally meaningful orders can be defined on intervals. Boost provides these various operators through namespace selection, the other libraries provide them through calls to explicitly-named functions. However, by implementing these operators so that they return more than just a boolean, no named functions are needed in this proposal to express “certain” and “possible” interval comparisons.

The Fortran community is also very active in the domain of interval arithmetic and this proposal is on par with a proposal[4)] that was written for this language. In the case of C++ though, interval arithmetic does not require any language changes and can (should) be implemented in a library.

We now raise a few more specialized points about the present proposal, and their rationales:

— **Why support several number types as template parameter?**
It is not much harder to support all three built-in floating point types `float|double|long double` than to support only one of them.

— **User defined types as template parameter?**
If some multiprecision integer or rational number type gets standardized, then it will be nice to be able to plug them in `interval`, and use some of the functions. In the mean time, this proposal does not address the support of UDTs as template parameter, as it would involve finding a way to specify a rounding mode or something equivalent, which gets complicated. The Boost.Interval library supports this.

— **Integral types as template parameter?**
We believe that there are not enough use-cases for this to be standardized. One notable exception is for the boolean case, with `interval<bool>` serving as the natural return type for interval comparisons.

— **Support for decimal floating types?**
There is a proposal for adding decimal floating point types to the standard library (N1776, Decimal types for C++, by Robert Klarer).
As they are rounded numeric types, it would make sense to support them, but it may be preferable to wait for a wide acceptance of the decimal types before envisioning decimal intervals.

— **Relation to existing standards : IEEE-754 and LIA-123?**
Run time choices of rounding modes are not part of the LIA standards, but are part of the IEEE 754 standard. The typical implementation of `interval` is probably going to rely on these, but the LIA standard should be enough to implement `interval` for floating point types.

4) `http://interval.louisiana.edu/F90/f96-pro.asc`

— **Precision of intervals**
We do not think it should be required that the smallest intervals preserving the inclusion property be returned by arithmetic operations, since this can be hard (or slow) to implement in LIA without IEEE 754 support. For these reasons, we propose that the standard should not require implementations to return the sharpest intervals possible, but only preserve the containment property for all functions. This issue then becomes a QOI issue, and there will be some leeway for implementers to choose between speed and precision.

— `inf() <= sup()` **invariant**
Typical uses of intervals require this property. There are some applications of intervals which use reversed order of the bounds `inf() > sup()` (modal intervals, or next items), but they are probably not worth taking into account in the standard.

— **Why no mutable versions of the access functions** `inf()` **and** `sup()`**?**
Because modifying the bounds needs to assert the invariant `inf()<=sup()`. To compensate, it would be possible to add functions like `void assign_inf(const T&)`, but it is not clear that this is really useful.

— **Division by an interval containing zero?**
A possible design choice is to allow intervals where `inf() > sup()` to represent the result of the division by an interval containing zero, as the union of two intervals $(-\infty,$`sup()`$]\cup[$`inf()`$,\infty)$, which is a tighter result than `interval::whole()`. We feel the better tightness does not, however, warrant the complications it introduces into the other arithmetic operations.

— **Empty interval**
The empty interval is useful for set-based operations, and it is crucial in making it possible to avoid exceptions altogether in the design. It can be made to play nicely with the arithmetic operations as well, so we decided to add support for it. It can be represented in several ways :
- either using NaNs for the bounds, which allows to propagate the empty intervals through arithmetic operations like addition without additional runtime cost.
- for implementations that do not support NaNs, the empty interval can be represented by reverse ordered bounds like `[1;0]`, at the cost of some checks at run time to distinguish the empty interval.
We propose that the standard does not specify which, giving the implementers more leeway to choose between complexity of implementation and efficiency.

Note that the result of `operator<<(interval<T>::empty())` is *not* unspecified, in fact, it is fixed to "[1;0]". This is also compatible with the constructor.

— **Why** `interval<T>::empty()` **and not** `is_empty(interval<T> const&)`**?**
In a previous version, we had a function template `is_empty(interval<T> const&)`, and then realized there was a conflict with the library TR1 template `is_empty` (part of the type traits). Instead of coming up with a different name, we decided to treat interval as a container for this purpose only, with a member function `bool empty() const;`.
In this same previous version, the function `interval<T>::empty()` was static and returned an empty interval (similar to `interval<T>::whole()`). To create and use an empty interval, one should simply use a default-constructed `interval<T> empty;`

— **What should the** `inf()` **and** `sup()` **of an empty interval return?**
We propose to leave it unspecified as well. Depending on the implementation of empty intervals, it could throw an exception, return a NaN or the actual bounds stored, or return $+\infty$ and $-\infty$ (at the cost of a test within `inf()`). All of these options have their advantages, in particular returning NaN would allow to bypass dynamic emptiness testing in most routines. If it matters for the users, they should wrap their calls to `inf()` and `sup()` with a non-emptiness test, but the basic access, especially for the internals of the interval class, should not be penalized by a mandatory test.

— **What is the behavior on out-of-range argument values? E.g., should** `sqrt(x)` **throw if interval** `x` **contains negative values?**
We decided to interpret the inclusion property as allowing negative values for `sqrt` without throwing

exception, simply saying that `sqrt(i)` contains `sqrt(x)` for any positive `x` in `i` (and is empty if `i` is entirely negative). Similarly for any function whose domain is a subset of the representable `T`. This is the most inclusive view since it allows for silent error propagation and recovery for `sqrt(i)` where `i` is (in theory) guaranteed to represent a positive number but its interval enclosure contains negative numbers.

Note that, it is always possible for the user to check manually the range of the arguments (e.g., does `i` contain negative numbers) and to choose the appropriate action. As Bill Walster puts it: "Depending on the context, an empty result of an arithmetic operation or function evaluation may or may not indicate that an error has been made. An exception-like feature is needed to provide a test for expression continuity, which is an assumption for the interval Newton algorithm and the Brouwer fixed-point theorem. This will only be needed in special circumstances and therefore needs to be invokable only when required."

— **Why use the notation $[x;y]$ in the documentation and in the I/O `operator<<`?**
Complex numbers already use the notation (x,y) and the use of brackets is customary for representing interval. We use the semicolon ';' instead of the comma ',' as the separator, to avoid conflicts with locales (as far as we know, none of them use ';' as a decimal point, but some do use the comma ',').

— **Does the I/O `operator<<` satisfy the inclusion property?**
Unfortunately, there is no way to express this in standardese, because I/O operators do not have a way of specifying rounding modes. We feel a good implementation should guarantee the inclusion property, but that has to be left to the implementer. Note that Sun's implementation does an excellent job with the I/O operators with respect to the inclusion property.

— **Specializations for `numeric_limits< interval<T> >`?**
We do not see a meaningful specialization of `numeric_limits` for `interval`.

— **Why allow** `interval<bool>`?
The choice has been made to introduce `interval<bool>`, which is similar in spirit to Boost.Tribool, except that it throws an exception for invalid conversions (à la `dynamic_cast`), and also supports the empty interval. Note that the conversion to `bool` *should* throw for an empty or indeterminate `interval<bool>`, but it is possible for the user to test for these cases by hand and avoid exceptions altogether.
It is primarily used for the result of comparisons (see next item), because we wanted to avoid systematically throwing an exception when the comparison returns an empty or indeterminate interval.
Another reason to allow `interval<bool>` is that it can be useful for fuzzy computations, independent of any numerical context.

— **Behavior of the comparison between empty or overlapping intervals?**
The overall design of comparisons is in line with the inclusion property, and hence mathematically consistent and easy to use: the comparison of two overlapping intervals is neither `true` nor `false`, but could be either one, hence the result can be naturally described as the interval `[false,true]`, also called *indeterminate*. Similarly, when comparing with an empty interval, the result is empty.
The choice of not returning `bool` allows users to decide if they wants to take the penalty of an exception or not. Exceptions can be useful, because there is no better way to treat the result when used in an expression that expects a `bool` (e.g., a conditional statement). Nevertheless, comparisons of overlapping intervals might be frequent in some applications; in that case, the result can be examined by the user without a conversion to `bool` and processed appropriately.

— **Should `std::set<interval>` work?**
Similarly to Library Issue 388 concerning `std::set<std::complex>`, we decided to make it illegal (see 26.6.7, clause 2, below). In order to safely use `std::set<interval>`, users have to provide a functor themselves (which could implement lexicographic order for example).
The problem is that the only natural strict weak ordering on `interval<T>`, which we provide as `operator<(interval,interval)`. is the one that extends the order on `T` when the intervals are disjoint, and that its use in set will throw when the two intervals overlap.

Note that, as long as none of the intervals in the set overlap, the set will work. Hence it is allowed for the programmers to use `std::set<interval>` at their own risk, as long as they can guarantee that none of the intervals overlap (by opposition to `std::set<std::complex>` which will not compile). We do not see any reason to disallow this possibility. But as far as we understand it, nothing in the standard describes the behavior of `std::set` when the comparison throws (even if the exception is caught right outside the call to `set::insert()`; it could for instance exhibit a memory leak if it creates a node before doing all the comparisons).

— **Should std::valarray<interval> be allowed?**
We do not see any reason why not.

— **Optimization expectations?**
One goal of the standardization of interval arithmetic is to make an implementation close to compilers, hence motivate some optimization work. These are mostly QOI issues, but we would like to mention two optimizations that we think are important to keep in mind when designing this proposal: rounding mode changes are costly for basic operations like interval addition/multiplication. And efficiency of these operations is important. Fortunately, there are tricks to eliminate most rounding mode changes.
The first one is the observation that the addition `a+b` rounded towards minus infinity is the same as `-(-a-b)` with operations rounded towards plus infinity, so that the same rounding mode can be used for both the lower and upper bounds. The same trick can be applied to many other operations. Care has to be taken, though, for machines where double rounding can have an effect (e.g. x86).
The second is the hope that the compiler, provided it has some knowledge of rounding mode change functions, can eliminate a rounding mode change if it knows that it is the same as the current one (previously changed). It could also eliminate consecutive rounding mode changes, provided that there is no floating point operation in between that can be affected.

IV Proposed Text for the Standard

In Chapter 26, Numerics library.

Add `interval` to paragraph 2, and change Table 79 to :

Table 79—Numerics library summary

Subclause	Header(s)
26.1 Requirements	
26.2 Complex numbers	`<complex>`
26.3 Numeric arrays	`<valarray>`
26.4 Generalized numeric operations	`<numeric>`
26.5 C library	`<cmath>` `<cstdlib>`
26.6 Interval arithmetic	`<interval>`

In 26.1, change paragraph 1 to add `interval`.

Change footnote 253 to add `interval` as allowed parameter to `valarray`.

Addition of the following section 26.6 :

26.6 Interval numbers [lib.interval.numbers]

1 The header `<interval>` defines a class template, and numerous functions for representing and manipulating numerical intervals.

2 The effect of instantiating the template `interval` for any type other than `float`, `double`, `long double`, or `bool` is unspecified.

3 Interval arithmetic is a basic tool for certified mathematical computations. The most important property is the *inclusion property*, which states that the result of the extension of a function over an interval must contain all the results of that function for all the values over this interval. This applies to the elementary arithmetic operations as well, in the sense that the interval resulting from an arithmetic operation over any number of intervals is guaranteed to contain any result of the operation where the operands hold any real values taken in the interval operand(s).

4 This notion is also extended to booleans to represent the return values of comparisons of intervals. The inclusion property is similarly used for the specification of the extension of boolean functions to boolean intervals. The `interval<bool>(false,true)` is used to represent an unknown boolean and is named an *indeterminate*. A conversion from `interval<bool>` to `bool` is also defined, which throws an exception if the boolean interval is not a singleton boolean value.

26.6.1 Header `<interval>` synopsis [lib.interval.synopsis]

```
namespace std {

// forward declarations:
template <class T> class interval;
template <> class interval<float>;
template <> class interval<double>;
template <> class interval<long double>;
template <> class interval<bool>;

// arithmetic operators:
template<class T> interval<T> operator+(const interval<T>&);
template<class T> interval<T> operator+(const interval<T>&, const interval<T>&);
template<class T> interval<T> operator+(const interval<T>&, const T&);
template<class T> interval<T> operator+(const T&, const interval<T>&);
template<class T> interval<T> operator-(const interval<T>&);
template<class T> interval<T> operator-(const interval<T>&, const interval<T>&);
template<class T> interval<T> operator-(const interval<T>&, const T&);
template<class T> interval<T> operator-(const T&, const interval<T>&);
template<class T> interval<T> operator*(const interval<T>&, const interval<T>&);
template<class T> interval<T> operator*(const interval<T>&, const T&);
template<class T> interval<T> operator*(const T&, const interval<T>&);
template<class T> interval<T> operator/(const interval<T>&, const interval<T>&);
template<class T> interval<T> operator/(const interval<T>&, const T&);
template<class T> interval<T> operator/(const T&, const interval<T>&);

// comparison operators:
template<class T> interval<bool> operator==(const interval<T>&, const interval<T>&);
template<class T> interval<bool> operator==(const interval<T>&, const T&);
template<class T> interval<bool> operator==(const T&, const interval<T>&);
template<class T> interval<bool> operator!=(const interval<T>&, const interval<T>&);
template<class T> interval<bool> operator!=(const interval<T>&, const T&);
template<class T> interval<bool> operator!=(const T&, const interval<T>&);
template<class T> interval<bool> operator<(const interval<T>&, const interval<T>&);
template<class T> interval<bool> operator<(const interval<T>&, const T&);
template<class T> interval<bool> operator<(const T&, const interval<T>&);
template<class T> interval<bool> operator>(const interval<T>&, const interval<T>&);
template<class T> interval<bool> operator>(const interval<T>&, const T&);
template<class T> interval<bool> operator>(const T&, const interval<T>&);
template<class T> interval<bool> operator<=(const interval<T>&, const interval<T>&);
template<class T> interval<bool> operator<=(const interval<T>&, const T&);
template<class T> interval<bool> operator<=(const T&, const interval<T>&);
template<class T> interval<bool> operator>=(const interval<T>&, const interval<T>&);
template<class T> interval<bool> operator>=(const interval<T>&, const T&);
template<class T> interval<bool> operator>=(const T&, const interval<T>&);

// stream operators:
template<class T, class charT, class traits>
        basic_istream<charT, traits>&
        operator>>(basic_istream<charT, traits>&, interval<T>&);
template<class T, class charT, class traits>
        basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>&, const interval<T>&);

// values:
template<class T> T inf(const interval<T>&);
template<class T> T sup(const interval<T>&);
template<class T> T midpoint(const interval<T>&);
template<class T> T width(const interval<T>&);
template<class T> interval<T> abs(const interval<T>&);
template<class T> interval<T> square(const interval<T>&);

// algebraic operators:
template<class T> interval<T> sqrt(const interval<T>&);

// set operations:
template<class T> bool is_singleton(const interval<T>&);
template<class T> bool contains(const interval<T>&, const T&);
template<class T> bool contains(const interval<T>&, const interval<T>&);
template<class T> bool overlap(const interval<T>&, const interval<T>&);
template<class T> interval<T> intersect(const interval<T>&, const interval<T>&);
template<class T> interval<T> hull(const interval<T>&, const interval<T>&);
template<class T> std::pair<interval<T>, interval<T> >
        split(const interval<T>&, const T&);
template<class T> std::pair<interval<T>, interval<T> >
        bisect(const interval<T>&);

}  // of namespace std
```

26.6.2 `interval` class template [lib.interval]

```
namespace std {

template < class T >
class interval
{
public:
  typedef T  value_type;

  interval();
  interval(const T& t);
  interval(const T& lo, const T& hi);
  template <class X> explicit interval(const X&);
  template <class X> interval(const X&, const X&);
  template <class X> interval(const interval<X>&);

  bool empty() const;

  T inf() const;
  T sup() const;
  interval& operator=(const T&);
  interval& operator+=(const T&);
  interval& operator-=(const T&);
  interval& operator*=(const T&);
  interval& operator/=(const T&);

  interval& operator=(const interval&);
  template <class X> interval& operator=(const interval<X>&);
  template <class X> interval& operator+=(const interval<X>&);
  template <class X> interval& operator-=(const interval<X>&);
  template <class X> interval& operator*=(const interval<X>&);
  template <class X> interval& operator/=(const interval<X>&);

  static interval whole();
};

}  // of namespace std
```

26.6.3 `interval` numeric specializations [lib.interval.special]

```
namespace std {

template <> class interval<float>
{
public:
  typedef float  value_type;

  interval();
  interval(const float& t);
  interval(const float& lo, const float& hi);
  explicit interval(const double & t);
  interval(const double & lo, const double & hi);
  explicit interval(const long double & t);
  interval(const long double & lo, const long double & hi);

  interval(const interval&);
  explicit interval(const interval<double>&);
  explicit interval(const interval<long double>&);

  bool empty() const;

  float inf() const;
  float sup() const;

  interval<float>& operator=(const float&);
  interval<float>& operator+=(const float&);
  interval<float>& operator-=(const float&);
  interval<float>& operator*=(const float&);
  interval<float>& operator/=(const float&);

  interval<float>& operator=(const interval<float>&);
  template <class T> interval<float>& operator=(const interval<T>&);
  template <class T> interval<float>& operator+=(const interval<T>&);
  template <class T> interval<float>& operator-=(const interval<T>&);
  template <class T> interval<float>& operator*=(const interval<T>&);
  template <class T> interval<float>& operator/=(const interval<T>&);
  static interval<float> whole();
};

template <> class interval<double>
{
public:
  typedef double  value_type;

  interval();
  interval(const double& t);
  interval(const double& lo, const double& hi);
  explicit interval(const long double & t);
  interval(const long double & lo, const long double & hi);

  interval(const interval<float>&);
  explicit interval(const interval<long double>&);

  bool empty() const;

  double inf() const;
  double sup() const;

  interval<double>& operator=(const double&);
  interval<double>& operator+=(const double&);
  interval<double>& operator-=(const double&);
  interval<double>& operator*=(const double&);
  interval<double>& operator/=(const double&);

  interval<double>& operator=(const interval<double>&);
  template <class T> interval<double>& operator=(const interval<T>&);
  template <class T> interval<double>& operator+=(const interval<T>&);
  template <class T> interval<double>& operator-=(const interval<T>&);
  template <class T> interval<double>& operator*=(const interval<T>&);
  template <class T> interval<double>& operator/=(const interval<T>&);

  static interval<double> whole();
};

template <> class interval<long double>
{
public:
  typedef long double  value_type;

  interval();
  interval(const long double& t);
  interval(const long double& lo, const long double& hi);

  interval(const interval<float>&);
  interval(const interval<double>&);

  bool empty() const;

  long double inf() const;
  long double sup() const;

  interval<long double>& operator=(const long double&);
  interval<long double>& operator+=(const long double&);
  interval<long double>& operator-=(const long double&);
  interval<long double>& operator*=(const long double&);
  interval<long double>& operator/=(const long double&);

  interval<long double>& operator=(const interval<long double>&);
  template <class T> interval<long double>& operator=(const interval<T>&);
  template <class T> interval<long double>& operator+=(const interval<T>&);
  template <class T> interval<long double>& operator-=(const interval<T>&);
  template <class T> interval<long double>& operator*=(const interval<T>&);
  template <class T> interval<long double>& operator/=(const interval<T>&);

  static interval<long double> whole();
};

} // of namespace std
```

26.6.4 `interval` member functions [lib.interval.members]

```
  template<class T> interval();
```

1 **Effects:** Constructs an empty interval.
2 **Postcondition:** `this->empty()` is true.

```
  template<class T> interval(const T& t);
```

3 **Effects:** Constructs a singleton interval $[t;t]$.
4 **Postcondition:** `inf() == t && sup() == t`.
5 **Notes:** Undefined if `t` is not a finite number.

```
  template<class T> interval(const T& lo, const T& hi);
```

6 **Effects:** Constructs an interval $[lo;hi]$ if $lo \leq hi$, otherwise an empty interval.
7 **Postcondition:** `inf() == lo && sup() == hi` if $lo \leq hi$, otherwise `this->empty()` is true.
8 **Notes:** Undefined if `lo` or `hi` is not a finite number.

```
  template<class T> template < class X > interval<T>(const X& x);
```

9 **Effects:** Constructs an interval containing x.
10 **Postcondition:** `inf() <= x && x <= sup()`.
11 **Notes:** Undefined if `x` is not a finite number.

```
  template<class T> template < class X > interval<T>(const X& lo, const X& hi);
```

12 **Effects:** Constructs an `interval<T>` containing $[lo;hi]$ if $lo \leq hi$, otherwise an empty interval.
13 **Postcondition:** `inf() <= lo && hi <= sup()` if $lo \leq hi$, otherwise `this->empty()` is true.
14 **Notes:** Undefined if `lo` or `hi` is not a finite number.

```
  template<class T> template < class X > interval<T>(const interval<X>& i);
```

15 **Effects:** Construct an `interval<T>` containing i.
16 **Postcondition:** `inf() <= i.inf() && i.sup() <= sup()`

```
template<class T> bool empty() const;
```

17 **Returns:** `true` if `*this` is empty.
18 **Notes:** `empty()` shall not be true if `x.inf() <= x.sup()`.

```
template<class T> T inf() const;
```

19 **Returns:** The lower bound of `*this`.
20 **Note:** Undefined if `*this` is empty.

```
template<class T> T sup() const;
```

21 **Returns:** The upper bound of `*this`.
22 **Note:** Undefined if `*this` is empty.

26.6.5 `interval` member operators [lib.interval.members.ops]

```
template<class T> interval<T>& operator+=(const T& rhs);
```

1 **Returns:** `*this += interval<T>(rhs)`.

```
template<class T> interval<T>& operator-=(const T& rhs);
```

2 **Returns:** `*this -= interval<T>(rhs)`.

```
template<class T> interval<T>& operator*=(const T& rhs);
```

3 **Returns:** `*this *= interval<T>(rhs)`.

```
template<class T> interval<T>& operator/=(const T& rhs);
```

4 **Returns:** `*this /= interval<T>(rhs)`.

```
template<class T> interval<T>& operator+=(const interval<T>& rhs);
```

5 **Effects:** Adds the interval value `rhs` to the interval value `*this`, and stores the result in `*this`.
6 **Returns:** `*this`.
7 **Postcondition:** If the value lhs of `*this` prior to the addition is non empty, `*this` contains $[xl+yl, xu+yu]$ where $lhs = [xl;xu]$ and $rhs = [yl;yu]$ and all operations are computed exactly, and `this->empty()` is true otherwise.

```
template<class T> interval<T>& operator-=(const interval<T>& rhs);
```

8 **Effects:** Subtracts the interval value `rhs` from the interval `*this`, and stores the result in `*this`.
9 **Returns:** `*this`.
10 **Postcondition:** If the value lhs of `*this` prior to the subtraction is non empty, `*this` contains $[xl-yu, xu-yl]$ where $lhs = [xl;xu]$ and $rhs = [yl;yu]$ and all operations are computed exactly, and `this->empty()` is true otherwise.

```
template<class T> interval<T>& operator*=(const interval<T>& rhs);
```

11 **Effects:** Multiplies the interval `*this` by the interval value `rhs`, and stores the result in `*this`.
12 **Returns:** `*this`.
13 **Postcondition:** If the value *lhs* of `*this` prior to the multiplication is non empty, `*this` contains $[\min(xl * yl, xl * yu, xu * yl, xu * yu); \max(xl * yl, xl * yu, xu * yl, xu * yu)]$ where $lhs = [xl; xu]$ and $rhs = [yl; yu]$ and all operations are computed exactly, and `this->empty()` is true otherwise.

```
  template<class T> interval<T>& operator/=(const interval<T>& rhs);
```

14 **Effects:** Stores an empty `interval<T>` in `*this` if `rhs` is empty or the singleton `interval<T>(T(0))`, otherwise does not change `*this` if it already contains `interval<T>(T(0))` otherwise stores `interval<T>::whole()` in `*this` if `rhs` strictly contains `T(0)`, otherwise divides the interval `*this` by the interval value `rhs` and stores the result in `*this`.
15 **Returns:** `*this`.
16 **Postcondition:** If the value *lhs* of `*this` prior to the multiplication is non empty and if `rhs` does not strictly contain `T(0)`, `*this` contains $[\min(xl/yl, xl/yu, xu/yl, xu/yu); \max(xl/yl, xl/yu, xu/yl, xu/yu)]$ where $lhs = [xl; xu]$ and $rhs = [yl; yu]$, all operations are computed exactly.
17 **Note:** If only one bound of `rhs` is zero and `*this` does not contain both negative and positive values, the stored interval shall not contain values that are of an unexpected sign. In particular it shall not be `interval<T>::whole()`.

26.6.6 `interval` non-member operations [lib.interval.ops]

```
template<class T> interval<T> operator+(const interval<T>& x);
```

1 **Notes:** Unary operator
2 **Returns:** `interval<T>(x)`

```
template<class T> interval<T> operator+(const interval<T>& lhs, const interval<T>& rhs);
template<class T> interval<T> operator+(const interval<T>& lhs, const T& rhs);
template<class T> interval<T> operator+(const T& lhs, const interval<T>& rhs);
```

3 **Returns:** `interval<T>(lhs) += rhs`.

```
template<class T> interval<T> operator-(const interval<T>& x);
```

4 **Notes:** Unary operator.
5 **Returns:** the opposite of the interval `x`.

```
template<class T> interval<T> operator-(const interval<T>& lhs, const interval<T>& rhs);
template<class T> interval<T> operator-(const interval<T>& lhs, const T& rhs);
template<class T> interval<T> operator-(const T& lhs, const interval<T>& rhs );
```

6 **Returns:** `interval<T>(lhs) -= rhs`.

```
template<class T> interval<T> operator*(const interval<T>& lhs, const interval<T>& rhs);
template<class T> interval<T> operator*(const interval<T>& lhs, const T& rhs);
template<class T> interval<T> operator*(const T& lhs, const interval<T>& rhs);
```

7 **Returns:** `interval<T>(lhs) *= rhs`.

```
template<class T> interval<T> operator/(const interval<T>& lhs, const interval<T>& rhs);
template<class T> interval<T> operator/(const interval<T>& lhs, const T& rhs);
template<class T> interval<T> operator/(const T& lhs, const interval<T>& rhs);
```

8 **Returns:** `interval<T>(lhs) /= rhs`.

26.6.7 `interval` comparisons [lib.interval.comps]

1 The equality and relational comparison operators on intervals are pure interval extensions of the corresponding comparison operators on the value type, using the inclusion property, and thus return an object of type `interval<bool>`. That way, comparison operators return `true` when the comparison is true for all pairs of values taken in the arguments, `false` when the comparison is false for all such pairs, an empty `interval<bool>` when one or two arguments are empty, and `interval<bool>::indeterminate()` otherwise.

2 The order defined on `interval<T>` by `operator<` is *not* a strict weak ordering as defined in 25.3 [lib.alg.sorting] due to the possible throwing of an exception when converting `interval<bool>` to `bool`.

```
template<class T> interval<bool> operator==(const interval<T>& lhs, const interval<T>& rhs);
template<class T> interval<bool> operator==(const interval<T>& lhs, const T& rhs);
template<class T> interval<bool> operator==(const T& lhs, const interval<T>& rhs);
```

3 **Returns:** an empty `interval<bool>` if either or both of `lhs` or `rhs` is empty, `false` if `lhs` and `rhs` are disjoint, `true` if both are the same singleton, and `interval<bool>::indeterminate()` otherwise.
4 **Notes:** The *T* arguments *lhs* or *rhs* are implicitely converted to `interval<T>(lhs)` and `interval<T>(rhs)`. The result is undefined if they are not finite numbers.

```
template<class T> interval<bool> operator!=(const interval<T>& lhs, const interval<T>& rhs);
template<class T> interval<bool> operator!=(const interval<T>& lhs, const T& rhs);
template<class T> interval<bool> operator!=(const T& lhs, const interval<T>& rhs);
```

5 **Returns:** `!( lhs == rhs )`.

```
template<class T> interval<bool> operator<(const interval<T>& lhs, const interval<T>& rhs);
template<class T> interval<bool> operator<(const interval<T>& lhs, const T& rhs);
template<class T> interval<bool> operator<(const T&lhs, const interval<T>& rhs);
```

6 **Returns:** an empty `interval<bool>` if either or both of `lhs` or `rhs` is empty, `true` if `lhs.sup() < rhs.inf()` is true, `false` if `lhs.inf() <= rhs.sup()` is true, and `interval<bool>::indeterminate()` otherwise.
7 **Notes:** The *T* arguments *lhs* or *rhs* are implicitely converted to `interval<T>(lhs)` and `interval<T>(rhs)`. The result is undefined if they are not finite numbers.

```
template<class T> interval<bool> operator<=(const interval<T>& lhs, const interval<T>& rhs);
template<class T> interval<bool> operator<=(const interval<T>& lhs, const T& rhs);
template<class T> interval<bool> operator<=(const T& lhs, const interval<T>& rhs);
```

8 **Returns:** `!( rhs < lhs )`.

```
template<class T> interval<bool> operator>(const interval<T>& lhs, const interval<T>& rhs);
template<class T> interval<bool> operator>(const interval<T>& lhs, const T& rhs);
template<class T> interval<bool> operator>(const T& lhs, const interval<T>& rhs);
```

9 **Returns:** `rhs < lhs`.

```
template<class T> interval<bool> operator>=(const interval<T>& lhs, const interval<T>& rhs);
template<class T> interval<bool> operator>=(const interval<T>& lhs, const T& rhs);
template<class T> interval<bool> operator>=(const T& lhs, const interval<T>& rhs);
```

10 **Returns:** `!( lhs < rhs )`.

26.6.8 `interval` IO operations [lib.interval.io]

```
template<class T, class charT, class traits>
        basic_istream<charT, traits>&
        operator>>(basic_istream<charT, traits>& is, interval<T>& i);
```

1 **Effects:** Extracts an interval *i* of the form: *t*, [*u*], or [*u*;*v*], where *u* is the lower bound and v is the upper bound.
2 **Requires:** The input values be convertible to `T`.
If bad input is encountered, calls `is.setstate(ios::failbit)` (which may throw ios::failure 27.4.4.3).
3 **Returns:** `is`
4 **Notes:** This extraction is performed as a series of simpler extractions. Therefore, the skipping of whitespace is specified to be the same for each of the simpler extractions.

```
template<class T, class charT, class traits>
        basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os, const interval<T>& i);
```

5 **Effects:** Inserts the interval *i* onto the stream *os* as if it were implemented as follows:

```
template<class T, class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const interval<T>& i)
{
  basic_ostringstream<charT, traits> s;
  if (i.empty())
    s << "[1;0]";
  else {
    s.flags(os.flags());
    s.imbue(os.getloc());
    s.precision(os.precision());
    s << "[" << i.inf() << ";" << i.sup() << "]";
  }
  return os << s.str();
}
```

26.6.9 `interval` value operations [lib.interval.value.ops]

```
template<class T> T inf(const interval<T>& x);
```

1 **Returns:** `x.inf()`.

```
template<class T> T sup(const interval<T>& x);
```

2 **Returns:** `x.sup()`.

```
template<class T> T midpoint(const interval<T>& x);
```

3 **Returns:** `(x.inf()+x.sup())/T(2.0)` when `x` is not empty, an implementation-defined value otherwise.

```
template<class T> T width(const interval<T>& x);
```

4 **Returns:** `T(0)` when `x` is a singleton, an implementation-defined value, neither positive nor 0, when `x` is empty, and an upper bound on `x.sup() - x.inf()` otherwise.

```
template<class T> interval<T> abs(const interval<T>& x);
```

5 **Returns:** `interval<T>( T(0), std::max(-x.inf(), x.sup()) )` if `x` contains `T(0)`, *x* if `x > T(0)`, and $-x$ if `x < T(0)`.

```
template<class T> interval<T> square(const interval<T>& x);
```

6 **Returns:** `abs(x * x)`.

26.6.10 `interval` algebraic operations [lib.interval.algebraic.ops]

```
template<class T> interval<T> sqrt(const interval<T>&x);
```

1 **Returns:** `interval<T>(std::sqrt(std::max(t.inf(),T(0))),std::sqrt(t.sup()))` if `!t.empty() && t.sup() >= T(0)`, and an empty `interval<T>` otherwise.

26.6.11 `interval` set operations [lib.interval.set.ops]

```
template<class T> bool is_singleton(const interval<T>& x);
```

1 **Returns:** `false` if `x` is empty, and `x.inf() == x.sup()` otherwise.

```
template<class T> bool equal(interval<T> const& x, interval<T> const& y);
```

2 **Returns:** true if both `x` and `y` are empty, true if neither `x` nor `y` is empty and `x.inf() == y.inf() && x.sup() == y.sup()` is true, and false otherwise.
3 **Notes:** Differs from `operator==` in the return type and the semantics (equality as set).

```
template<class T> bool contains(const interval<T>& lhs, const interval<T>& rhs);
```

4 **Returns:** true if `rhs` is empty, false if `lhs` is empty and `rhs` is not, `lhs.inf() <= rhs.inf() && rhs.sup() <= lhs.sup()` otherwise.

```
template<class T> bool contains(const interval<T>& lhs, const T& rhs);
```

5 **Returns:** `contains(lhs, interval<T>(rhs))`

```
template<class T> bool overlap(const interval<T>& x, const interval<T>& y);
```

6 **Returns:** true if neither `x` nor `y` is empty and `y.inf() <= x.sup() && x.inf() <= y.sup()` is true, false otherwise.

```
template<class T> bool comparable(const interval<T>& x, const interval<T>& y);
```

7 **Returns:** true if neither `x` nor `y` is empty and `x.sup() < y.inf() || y.sup() < x.inf()` is true, false otherwise.

```
template<class T> interval<T> intersect(const interval<T>& x, const interval<T>& y);
```

8 **Returns:** an empty `interval<T>` if `overlap(x, y)` is false, `interval<T>(std::max(x.inf(), y.inf()), std::min(x.sup(), y.sup()))` otherwise.

```
template<class T> interval<T> hull(const interval<T>& x, const interval<T>& y);
```

9 **Returns:** `x` if `y` is empty, `y` if `x` is empty, `interval<T>(std::min(x.inf(), y.inf()), std::max(x.sup(), y.sup()))` otherwise.

```
template<class T> std::pair<interval<T>, interval<T> >
        split(const interval<T>& x, const T& t);
```

10 **Returns:** a pair of intervals such that the first (resp. second) member is the smallest interval containing all values of `x` smaller (resp. larger) than or equal to `t`.

11 **Notes:** Returns a pair of two empty `interval<T>` if `x` is empty. Otherwise, the first member will be empty if and only if `t < inf(x)`, and the second member will be empty if and only if `sup(x) < t`. Undefined if `t` is not a finite number.

```
template<class T> std::pair<interval<T>, interval<T> >
        bisect(const interval<T>& x);
```

12 **Returns:** `split(x, midpoint(x))` if `x` is not empty, a pair of two empty `interval<T>` otherwise.

26.6.12 `interval` static value operations [lib.interval.static.value.ops]

```
  static interval<T> whole();
```

1 **Returns:** `interval<T>(-std::numeric_limits<T>::infinity(),std::numeric_limits<T>::infinity())`.

2 **Requires:** `std::numeric_limits<T>::has_infinity`.

26.6.13 `interval<bool>` specialization [lib.interval.bool]

In order to correctly handle return types of comparisons, we also allow a Boolean specialization of the `interval` class template. Note that the interface is intentionally different, since this is not a *numeric* specialization (and is not intended to be used for arithmetic computations).

```
namespace std {

// values:
bool inf(interval<bool> const&);
bool sup(interval<bool> const&);

// set operations:
bool is_indeterminate(interval<bool> const&);

// boolean operations:
interval<bool> operator!(interval<bool> const&);
interval<bool> operator==(bool, interval<bool> const&);
interval<bool> operator==(interval<bool> const&, bool);
interval<bool> operator==(interval<bool> const&, interval<bool> const&);

interval<bool> operator!=(bool, interval<bool> const&);
interval<bool> operator!=(interval<bool> const&, bool);
interval<bool> operator!=(interval<bool> const&, interval<bool> const&);

interval<bool> operator||(bool, interval<bool> const&);
interval<bool> operator||(interval<bool> const&, bool);
interval<bool> operator||(interval<bool> const&, interval<bool> const&);

interval<bool> operator&&(bool, interval<bool> const&);
interval<bool> operator&&(interval<bool> const&, bool);
interval<bool> operator&&(interval<bool> const&, interval<bool> const&);

template <>
class interval<bool>
{
public:
  typedef bool  value_type;

  interval();
  interval(bool t);
  interval(bool lo, bool hi);

  bool empty() const;

  bool inf() const;
  bool sup() const;

  operator bool() const;
  static interval indeterminate();
};

} // of namespace std
```

26.6.14 `interval<bool>` member functions [lib.interval.bool.members]

```
  interval();
```

1 **Effects:** Constructs an indeterminate interval.
2 **Postcondition:** `inf() == false && sup() == true`.

```
  interval(bool b);
```

3 **Effects:** Constructs a singleton interval $[b;b]$.
4 **Postcondition:** `inf() == b && sup() == b`.

```
  interval(bool lo, bool hi);
```

5 **Effects:** Constructs an interval $[lo;hi]$.
6 **Postcondition:** `inf() == lo && sup() == hi`.

```
bool empty() const;
```

7 **Returns:** `x.inf()==true && x.sup()==false`.

```
bool inf() const;
```

8 **Returns:** The lower bound of `*this`.

```
bool sup() const;
```

9 **Returns:** The upper bound of `*this`.

26.6.15 `interval<bool>` conversion to `bool` [lib.interval.bool.conversion]

```
operator bool() const;
```

1 **Effects:** Returns *b* if *i* is a singleton $[b;b]$.
2 **Throws:** `std::range_error()` otherwise.

26.6.16 `interval<bool>` non-member operations [lib.interval.bool.ops]

```
bool inf(const interval<bool>& x);
```

1 **Returns:** `x.inf()`.

```
bool sup(const interval<bool>&x);
```

2 **Returns:** `x.sup()`.

```
bool is_indeterminate(const interval<bool>& x);
```

3 **Returns:** `x.inf()==false && x.sup()==true`.

```
interval<bool> operator!(const interval<bool>& x);
```

4 **Returns:** `x` if `x` is empty or indeterminate, and `!bool(x)` otherwise.

```
interval<bool> operator==(const interval<bool>& lhs, const interval<bool>& rhs);
interval<bool> operator==(const interval<bool>& lhs, bool rhs);
interval<bool> operator==(bool lhs, const interval<bool>& rhs);
```

5 **Returns:** an empty `interval<bool>` if either or both of `lhs` or `rhs` is empty, `interval<bool>::indeterminate()` if either or both of `lhs` or `rhs` is indeterminate, and `bool(lhs) == bool(rhs)` otherwise.

```
interval<bool> operator!=(const interval<bool>& lhs, const interval<bool>& rhs);
interval<bool> operator!=(const interval<bool>& lhs, bool rhs);
interval<bool> operator!=(bool lhs, const interval<bool>& rhs);
```

6 **Returns:** `!( lhs == rhs )`

```
interval<bool> operator||(const interval<bool>& lhs, const interval<bool>& rhs);
interval<bool> operator||(const interval<bool>& lhs, bool rhs);
interval<bool> operator||(bool lhs, const interval<bool>& rhs);
```

7 **Returns:** an empty `interval<bool>` if either or both of `lhs` or `rhs` is empty, and `interval<bool>(a.inf() || b.inf(), a.sup() || b.sup())` otherwise.

8 **Notes:** Always returns `true` if `lhs == true` or `rhs == true` and the other interval is not empty.

```
interval<bool> operator&&(const interval<bool>& lhs, const interval<bool>& rhs);
interval<bool> operator&&(const interval<bool>& lhs, bool rhs);
interval<bool> operator&&(bool lhs, const interval<bool>& rhs);
```

9 **Returns:** an empty `interval<bool>` if either or both of `lhs` or `rhs` is empty, and `interval<bool>(a.inf() && b.inf(), a.sup() && b.sup())` otherwise.

10 **Notes:** Always returns `false` if `lhs == false` or `rhs == false` and the other interval is not empty.

26.6.17 `interval<bool>` static value operations [lib.interval.bool.static.value.ops]

```
  static interval<bool> indeterminate();
```

1 **Returns:** `interval<bool>(false,true)`.

V Possible extensions

V.1 `<cmath>` functions

It would also be very useful to have the equivalent of the `<cmath>` functions, at least for some of these functions, and for floating point base types.

However, these functions (`cos`, `exp`...) are much harder to implement correctly, since their behavior with respect to rounding mode is not specified by the IEEE 754 standard, and hardware implementations vary.

We must note however, that there exist several libraries which provide the needed functionality: IBM Mathlib, Sun libmcr[5)], CRlibm[6)], and MPFR[7)].

So this is not out of reach. If we were to include it in the proposal, it would replace the section *algebraic operators* in the synopsis above by the following:

```
namespace std {

// algebraic and transcendental functions:
template<class T> interval<T> acos(const interval<T>&);
template<class T> interval<T> asin(const interval<T>&);
template<class T> interval<T> atan(const interval<T>&);
template<class T> interval<T> cos(const interval<T>&);
template<class T> interval<T> cosh(const interval<T>&);
template<class T> interval<T> exp(const interval<T>&);
template<class T> interval<T> log(const interval<T>&);
template<class T> interval<T> log10(const interval<T>&);
template<class T> interval<T> pow(const interval<T>&, int);
template<class T> interval<T> pow(const interval<T>&, const T&);
template<class T> interval<T> pow(const interval<T>&, const interval<T>&);
template<class T> interval<T> pow(const T&, const interval<T>&);
template<class T> interval<T> sin(const interval<T>&);
template<class T> interval<T> sinh(const interval<T>&);
template<class T> interval<T> sqrt(const interval<T>&);
```

5) http://www.sun.com/download/products.xml?id=41797765
6) http://lipforge.ens-lyon.fr/www/crlibm/
7) http://www.mpfr.org/

```
template<class T> interval<T> tan(const interval<T>&);
template<class T> interval<T> tanh(const interval<T>&);

} // of namespace std
```

VI Examples of usage of the `interval` class.

We show how to implement a solver-type application using intervals. We emphasize these are only proof-of-concepts and in no case more than toy demo programs. Other proof-of-concept programs which could be demonstrated here would include certified evaluation of boolean predicates (e.g., as used in exact geometric computing), interval extensions of Newton's method...

A prototype implementation of this proposal and some example programs can be found at `http://www-sop.inria.fr/geometrica/team/Sylvain.Pion/cxx/`.

VI.1 Unidimensional solver

As an example of the usefulness of our proposal, we show how to implement a very simple unidimensional algebraic solver. In fact, the function to solve is passed a function object, which must be able to process intervals.

```
// Returns a sorted set of intervals (sub-intervals of current), which might contain zeros of f.
// The dichotomy is stopped when the width of subintervals is <= precision.
template < class Function, class OutputIterator, class T >
OutputIterator
solve(Function f, OutputIterator oit,
      interval<T> const& current, T const& precision = 0)
{
  interval<T> y = f(current);   // Evaluate f() over current interval.

  // Short circuit if current does not contain a zero of f
  if (! contains(y, T(0))) return oit;

  // Stop the dichotomy if res is small enough (this prevents (most probably) useless work).
  // Also stop if we have reached the maximal precision.
  interval<T> eps( - std::numeric_limits<T>::min(), std::numeric_limits<T>::min());
  if (contains(eps, y) || width(current) <= precision) {
    *oit++ = current;
    return oit;
  }

  // Else, do the dichotomy recursively.
  std::pair<I, I> ip = bisect(current);

  // Stop if we can't dichotomize anymore.
  if (is_singleton(ip.first) || is_singleton(ip.second)) {
    *oit++ = current;
    return oit;
  }

  oit  = solve(f, oit, ip.first,  precision);
  return solve(f, oit, ip.second, precision);
}
```

This solver is wrapped in the example code submitted with this proposal using a driver that parses expressions (using Boost.spirit) and produces an output similar to the following output:

```
Type an expression of a variable t... or [q or Q] to quit
   (t*t-2)*(t-3)^2*(t-6)*t*t*(t+6)^2
enter the bounds of the interval over which to search for zeroes:
   -10 10
enter the precision with which to isolate the zeroes:
  0.00000001
Solved with 403 recursive calls (7 intervals before merging)
Solutions (if any) lie in :
[-6.0000000055879354477;-5.9999999962747097015]
[-1.4142135623842477798;-1.4142135530710220337]
[-9.3132257461547851562e-09;9.3132257461547851562e-09]
[1.4142135530710220337;1.4142135623842477798]
[2.9999999981373548508;3.0000000074505805969]
[5.9999999962747097015;6.0000000055879354477]
```

The functor passed to `solve` evaluates the expression tree built by the parser, either for a `double`, or for an interval.

VI.2 Multi-dimensional solver

As an illustration to the power and ease of extension of the method, let us show how to generalize the previous solver to solve a system of polynomial equations (an active area of research in robotics and applied numerics). Consider the system:

This system is fully constrained but admits seven solutions and a one-dimensional singular solution. We solve it using the generalized bisection method. Assume that `width` has been extended to vectors of interval (by taking max of width) and that `assign_box(r,epsilon)` assigns `epsilon` to every component of the vector `r`.

```
// Returns a set of block-intervals (sub-blocks of current), which might contain zeros of f.
// The dichotomy is stopped when the width of subintervals is <= precision.
template < class Function, class OutputIterator, class T >
OutputIterator
solve(Function f, OutputIterator oit,
      vector< interval<T> > const& current, T const& precision = 0)
{
  typedef interval<T> I;
  typedef vector<I> A; // vector<I> of dimension n
  typedef typename Function::result_type R; // vector<I> of dimension m

  R res = f(current); // Evaluate f() over current interval.

  // Short circuit if current does not contain a zero of f
  if (! contains_zero(res)) return oit;

  // Stop the dichotomy if res is small enough (this prevents (most probably) useless work).
  // Also stop if we have reached the maximal precision.
  R r(res); // initialize dimension in case R is a vector
  assign_box(r, I( -std::numeric_limits<T>::min(), -std::numeric_limits<T>::min() ) );
  if (contains(r, res) || width(current) <= precision) {
    *oit++ = current;
    return oit;
  }

  // Otherwise bisect along every dimension
  A begin(current), end(current);
  for (size_t s=0; s<current.size(); ++s) {
    std::pair<I,I> p = bisect(current[s]);
    begin[s] = p.first; end[s] = p.second;
    // Stop if we hit a singleton along any dimension
    if (is_singleton(begin[s]) || is_singleton(end[s])) {
      *oit++ = current;
      return oit;
    }
  }

  // Use binary enumeration of all the sub-boxes of current
  A it(begin);
  while (true)
  {
    // Solve recursively
    oit = solve(f, oit, it, precision);
    // Do the ++
    for (size_t s=0; s<current.size(); ++s)
    {
       if (inf(it[s]) >= sup(begin[s])) {
        if (s == current.size()-1) return oit; // done!
        else it[s] = begin[s]; //
      } else {
        it[s] = end[s];
        break;
      }
    }
  }
}
```

Again, this solver is wrapped in the example code submitted with this proposal using a driver that parses expressions (using Boost.spirit) and produces an output similar to the following output:

```
enter the number of variables: 2
enter the variable names
  variable 0: x
  variable 1: y
enter the number of equations of the system... or [q or Q] to quit
  2
Type 2 expressions of the variables ...
  expr: x*x + y*y - 4
    parsing succeeded
  expr: (x-1)*(x-1) + (y-1)*(y-1) -4
    parsing succeeded
enter the bounds of the interval box over which to search for zeroes:
  dim 0: -10 10
  dim 1: -10 10
enter the precision with which to isolate the zeroes:
  0.000000001
Solved with 633 recursive calls
Solutions (if any) lie in :
[1.8228756549069657922;1.8228756554890424013] [-0.8228756563039496541;-0.82287565572187304497
[1.8228756549069657922;1.8228756554890424013] [-0.82287565572187304497;-0.8228756551397964358
[1.8228756554890424013;1.8228756560711190104] [-0.82287565572187304497;-0.8228756551397964358
[-0.8228756563039496541;-0.82287565572187304497] [1.8228756549069657922;1.8228756554890424013
[-0.82287565572187304497;-0.82287565513979643583] [1.8228756549069657922;1.822875655489042401
[-0.82287565572187304497;-0.82287565513979643583] [1.8228756554890424013;1.822875656071119010
```

VII Acknowledgements

We are grateful to the Boost community for its support, and the deep peer review of the Boost.Interval library. Special thanks go to Jens Maurer for starting the first version of what became Boost.Interval, and to him and the reliable computing community for archiving the discussions and design decisions that greatly helped the preparation of this proposal.

References

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

[2] H. Brönnimann, G. Melquiond, and S. Pion. The design of the Boost interval arithmetic library. Accepted for publication in *Theoretical Computer Science*, Special Issue on Real Numbers and Computers (RNC5). Preprint available at `http://photon.poly.edu/~hbr/publi/boost-interval-rnc5/tcs.pdf`

[3] H. Brönnimann, G. Melquiond, and S. Pion. The Boost interval arithmetic library. `http://www.boost.org/libs/numeric/interval/doc/interval.htm`

[4] [CGAL] CGAL. *Computational Geometry Algorithms Library*. `http://www.cgal.org/`

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

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

[7] 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.

[8] 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/`

[9] 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`

[10] R. Baker Kearfott and V. Kreinovich (eds.) Applications of Interval Computations. Kluwer, 1996.

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

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

[13] *Gaol, Not Just Another Interval Library*. `http://www.sourceforge.net/projects/gaol/`

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

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