Doc No: N2137=06-0207

A Proposal to add Interval Arithmetic to the C++ Standard Library (revision 2)

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

2006-11-01

Contents

[*]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 History of changes to this document

Since revision 1 (N2067=06-0137) :

— Added `nth_root` and `nth_root_rel` functions.

— Removed explicit bound definition of `whole`, `hull`, and `intersect`.

— Fixed set-inclusion comparisons and detailed `bool_set` comparisons.

— Fixed definition of `midpoint`, `lower`, and `upper` with respect to infinities.

— Removed generic definition of `interval<T>`.

— Removed template parameter `U`.

— Fixed 26.6/2 and removed notion of bounds from 26.6/3.

— Added a note that empty intervals have to be propagated.

— The default constructor now produces an uninitialized interval.

— `interval::empty_set()` returns an empty interval.

— Renamed `interval::whole()` to `interval::whole_set()`.

— Turned `interval::empty()` into a free function named `is_empty_set()`.

— Added `is_positively_bounded`, `is_negatively_bounded`, and `is_bounded` free functions.

— Replaced `width` function by a `radius` function.

— Removed memory layout specification and `data` member.

— Added partial functions with explicit flag.

— Fixed `atanh` definition and added out-of-domain version.

Since initial version (N1843=05-0103) :

— Changed to pass-by-value for intervals.

— Added a more complete list of mathematical functions.

— Added a mathematical definition of intervals.

— Added a note that the precision of the operations is left as a quality of implementation issue.

— Modified `interval<T>(T)` to safely handle exceptional values.

— Added `interval(char const *)` constructor.

— Added `data` member for fixing memory layout (yet not fixing memory content).

— Changed `inf` and `sup` members to `lower` and `upper`.

— Added the “inclusion property” to input and output operators.

— Defined arithmetic operators and constructors by “enclosures”.

— Defined `inf(interval<T>)` and `sup(interval<T>)` for empty intervals.

— Allowed some infinite bounds for `interval<T>(T,T)`.

— `interval<bool>` renamed to `bool_set`, and moved to a separate proposal.

— `bool_set` boolean logical operators replaced by logical operators.

— Removed copy constructors and copy assignment operators from `interval`.

— Added namespaces for relational operators.

— Added forward and relational mathematical functions.

II 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. 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 list a few here, while noting that this illustrative list is in no way exhaustive[2]:

— Controlling 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).

— 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 change 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 IA?

— The functionality is needed in many areas;

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

— 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/`.

[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/`

III 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?

It is a pure extension to the standard library.

An efficient implementation of the proposal will rely on specific optimizations from the compiler (e.g., optimizing away redundant FPU rounding mode changes), which we describe in the next section. Those optimizations, however, are not compulsory for implementing the proposal.

This proposal refers to the proposal on `bool_set` (N2046), and some parts (comparison operators) depend on it.

IV Design Decisions

Why did you choose the specific design that you did? What alternatives did you consider, and what are the trade-offs? 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 introduces a single class template `interval<T>`, together with numerical functions which guarantee the inclusion property. Like `std::complex<T>`, we decided to support the three built-in floating-point types and leave the rest unspecified. We support empty intervals, on the ground that intervals are sets, and a set model without the empty set is as strange as a floating-point model without zero. 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 `bool_set` (see N2046) 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 `bool_set` 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. However, for functions which are partially defined on the reals, an overload is also provided allowing to collect the information of continuity and definedness of the function over the input, using an additional boolean flag argument (this design choice is still debated).

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 `bool_set` 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 implementation-defined, although the I/O representation is fixed in this proposal. (See the item below for a discussion of possible choices.) The radius and midpoint of an empty interval are also implementation-defined (we recommend a NaN on systems that support it).

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; this proposal provides 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 for a basic implementation to support all three built-in floating-point types `float|double|long double` than to support only one of them. A quality implementation will provide sharper results for floating-point types with higher precision.

— **User-defined types as template parameter?**

  If some multi-precision 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.

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

— **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 `bool_set` serving as the natural return type for interval comparisons.

— **Support for decimal floating-int types?**

There is a proposal for adding decimal floating-point types to the standard library (N1977, 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 : IEC-559, IEEE-754 and LIA-123?**

Run time choices of rounding modes are not part of the LIA standards, but are part of the IEC 559 standard (which is the same as IEEE 754). 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.

— **Precision of intervals**

We do not think it should be required that the smallest intervals preserving the inclusion property be returned by operations, since this can be hard (or slow) to implement. 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.

One exception to this is that empty intervals have to be propagated through functions which are extension of real functions. This is easy to implement, and we believe it can be implemented without a noticeable performance impact.

— `lower() <= upper()` **invariant**

Typical uses of intervals require this property. There are some applications of intervals which use reversed order of the bounds `lower() > upper()` (modal intervals, for instance; see also the next rationales), but their semantics are more involved and their specificity makes them less candidate for standardization. Moreover, reversed bounds may possibly clash with the representation of the empty interval (see “empty interval” below) on platforms that do not support NaNs.

— **Why no mutable versions of the access functions** `lower()` **and** `upper()`**?**

Modifying the bounds directly could break internal invariants of an implementation, like `lower() <= upper()`. To compensate, it would be possible to add functions like `assign_lower(T)`, but it is not clear that this is really useful.

— **Division by an interval containing zero?**

There usually are two ways of expressing the division on sets: X/Y. Either as the reciprocal of the multiplication: $\{z \mid \exists x \in X \exists y \in Y\ x = y \cdot z\}$. Or as the extension of the division: $\{z \mid \exists x \in X \exists y \in Y\ z = x/y\}$. As long as Y does not contain zero, both results are identical. Both expressions have their applications. The second expression gives tighter intervals and it cannot be emulated with the first one, while the first expression can be emulated with the second. So we chose to implement the second expression.

— **Wrap-around intervals?**

A possible design choice is to allow intervals where `lower() > upper()` to represent the result of the division by an interval containing zero, as the union of two intervals $(-\infty,$`upper()`$] \cup [$`lower()`$,\infty)$, which is a tighter result than `interval::whole_set()`. We feel the better tightness does not, however, warrant the complications it introduces into the other arithmetic operations, and that should this tightness be necessary, it could still be handled explicitly by the application.

— **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.

— **What should the `lower()` and `upper()` 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 `lower()`). 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 `lower()` and `upper()` 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. The free functions `inf(interval<T>)` and `sup(interval<T>)` are defined for empty intervals, however.

— **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(X)` contains $\sqrt{x}$ for any non-negative real number x in `X` (and is empty if `X` is entirely negative). Similarly for any other partial function. This is the most inclusive view since it allows for silent error propagation and recovery for `sqrt(X)` where `X` is (in theory) guaranteed to represent a non-negative number but its interval enclosure contains negative numbers.

— **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 intervals. 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 ','). Providing flexibility using a mechanism based on facets is something which might be a useful addition to the proposal.

— **Why do the I/O operators have to satisfy the inclusion property?**

Every arithmetic operation of this proposal satisfies the inclusion property. This allows to perform guaranteed computations. Interactions with the user should provide the same guarantees. If the user inputs 0.1, it should not be interpreted as a singleton interval that does not even contain this value. And reciprocally, the applications should not display or store $[0.13;0.13]$ when the interval is a singleton containing $\frac{1}{8}$, and a 2-decimal-place display has been specified.

— **Specializations for `numeric_limits< interval<T> >`?**

We do not see any meaningful specialization of `numeric_limits` for `interval`.

— **Why not propose default comparison operators but namespaces?**

There is no single model of comparison operators that is compatible with the numeric interpretation of intervals. Having default operators would probably result in failure of generic template codes, because some programs may expect a "possibly" semantic while some may expect a "certainly" semantic while some may not even be able to cope with the fact that any reasonable ordering on intervals is necessarily a partial ordering. As a consequence, in order to prevent any misuse of these comparison operators, they have been put into a namespace and the user will have to import the one that corresponds to the semantic the code needs. The user is also free to define new versions if other semantics are required, there will not be any ambiguity at compile time.

— **Why use `bool_set` as a result of comparisons?**

  The overall design of comparisons in `bool_set_ops` 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. Returning a `bool_set` value allows the comparison operators to cover all these cases.

— **Why propose other comparison models?**

  Depending on the application domain, the `bool_set` model for comparisons may not be the most suitable one. The "certainly" and "possibly" models may be more useful. They can be emulated by the `bool_set` model by using appropriate conversions to `bool` but this requires a cumbersome syntax and may induce a performance drop. The "set inclusion" model is provided for compatibility with a number of interval applications.

— **Why do "certainly" comparisons return `true` for empty intervals?**

  A mathematical property quantified on an empty set holds true. Moreover, this allows for the property $\text{certainly}(\mathrm{X} < \mathrm{Y})$ `==` `!`$\text{possibly}(X \geq Y)$ to hold true whatever the intervals X and Y.

— **Should `std::set<interval>` work?**

  Similarly to Library Issue 388 concerning `std::set<std::complex>`, we decided to make it illegal, as there is no natural strict weak ordering on intervals. In order to use `std::set<interval>`, users have to provide a functor themselves (which could implement lexicographic order for example). We do not want to specialize `std::less` for intervals, as this function would not match the behavior of `operator<`.

— **Should `std::valarray<interval>` be allowed?**

  We do not see any reason why not.

— **Should intervals be passed by value or by reference?**

  This is a debated issue. The best solution usually depends on ABIs and compilers. However, it seems best from the conceptual stand-point to pass intervals by value, since intervals are small objects. Moreover, passing by value removes potential aliasing issues. Note that we differ here from `std::complex`. Note that this point is moot when functions are inline, which is expected to be the case for many cases.

— **Discontinuity flag**

  Interval arithmetic is a powerful tool for finding roots, solving differential equations, and so on, as dedicated interval algorithms guarantee that no solutions were missed. The robustness from these algorithms usually comes from theorems like Brouwer's (a continuous function from a bounded interval box into itself has at least one fixed-point). These theorems are applied by computing the range of an iterated function on a given domain and checking that the range is indeed contained into the domain.

  Unfortunately, due to out-of-domain values or discontinuities, the function could be wrongly believed to have a fixed-point while it does not. To detect these false-positives, interval computations can raise a flag when input intervals are such that applying fixed-point theorems would lead to wrong results.

  These failure conditions can be split into two flags that could be raised separately: an out-of-domain flag and a discontinuity flag. We do not, however, know of any application that would benefit from this split semantic, so we are only proposing a single flag: it is raised when an out-of-domain or a discontinuity happens.

Throwing an exception instead of raising a flag would be unhelpful, as the computed range is still perfectly valid and may be smaller than the starting domain. The flag does not tell that the computed range is wrong (it is not), it only tells that the program cannot yet conclude on the existence of a fixed-point.

— **Why an `interval(char const *)` constructor?**

This constructor is not meant to be used for user input; the STL I/O stream machinery should be used instead. This constructor makes it possible to write interval literals in a program, e.g. `interval<float> Pi("[3.1415926,3.1415927]");`. Its string argument is parsed with the C locale to prevent unexpected interaction with the user locale.

— **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.

V 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` is unspecified.

3 Intervals are connected subsets of the set of real numbers. Which subsets are representable by `interval<T>` is implementation-defined. An implementation shall support at least the empty set $\emptyset$, the whole set of real numbers $\mathbb{R}$, and any singleton interval $\{x\}$ for x a real number representable by a floating-point number of type `T`.

4 Interval arithmetic is a basic tool for certified mathematical computations. The key 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).

5 The precision of the resulting intervals is left as a quality of implementation issue[5]. However, all functions, except set operations, taking the empty interval as argument must return the empty interval.

6 An interval object is always in one of the two following exclusive states: initialized or uninitialized. The default constructor produces uninitialized intervals. The state is propagated through assignments and copy constructors. The other constructors produce initialized intervals. Other operations on uninitialized intervals have undefined behavior, while operations on initialized intervals follow the described semantic.

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>;

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

// bool_set comparison operators:
namespace bool_set_ops {
template<class T> bool_set operator==(interval<T>, interval<T>);
```

[5] There can be a trade-off between the efficiency and the precision for some operations.

```
template<class T> bool_set operator==(interval<T>, T);
template<class T> bool_set operator==(T, interval<T>);
template<class T> bool_set operator!=(interval<T>, interval<T>);
template<class T> bool_set operator!=(interval<T>, T);
template<class T> bool_set operator!=(T, interval<T>);
template<class T> bool_set operator<(interval<T>, interval<T>);
template<class T> bool_set operator<(interval<T>, T);
template<class T> bool_set operator<(T, interval<T>);
template<class T> bool_set operator>(interval<T>, interval<T>);
template<class T> bool_set operator>(interval<T>, T);
template<class T> bool_set operator>(T, interval<T>);
template<class T> bool_set operator<=(interval<T>, interval<T>);
template<class T> bool_set operator<=(interval<T>, T);
template<class T> bool_set operator<=(T, interval<T>);
template<class T> bool_set operator>=(interval<T>, interval<T>);
template<class T> bool_set operator>=(interval<T>, T);
template<class T> bool_set operator>=(T, interval<T>);
}

// possibly comparison operators:
namespace possibly_ops {
template<class T> bool operator==(interval<T>, interval<T>);
template<class T> bool operator==(interval<T>, T);
template<class T> bool operator==(T, interval<T>);
template<class T> bool operator!=(interval<T>, interval<T>);
template<class T> bool operator!=(interval<T>, T);
template<class T> bool operator!=(T, interval<T>);
template<class T> bool operator<(interval<T>, interval<T>);
template<class T> bool operator<(interval<T>, T);
template<class T> bool operator<(T, interval<T>);
template<class T> bool operator>(interval<T>, interval<T>);
template<class T> bool operator>(interval<T>, T);
template<class T> bool operator>(T, interval<T>);
template<class T> bool operator<=(interval<T>, interval<T>);
template<class T> bool operator<=(interval<T>, T);
template<class T> bool operator<=(T, interval<T>);
template<class T> bool operator>=(interval<T>, interval<T>);
template<class T> bool operator>=(interval<T>, T);
template<class T> bool operator>=(T, interval<T>);
}

// certainly comparison operators:
namespace certainly_ops {
template<class T> bool operator==(interval<T>, interval<T>);
template<class T> bool operator==(interval<T>, T);
template<class T> bool operator==(T, interval<T>);
template<class T> bool operator!=(interval<T>, interval<T>);
template<class T> bool operator!=(interval<T>, T);
template<class T> bool operator!=(T, interval<T>);
template<class T> bool operator<(interval<T>, interval<T>);
template<class T> bool operator<(interval<T>, T);
template<class T> bool operator<(T, interval<T>);
template<class T> bool operator>(interval<T>, interval<T>);
template<class T> bool operator>(interval<T>, T);
template<class T> bool operator>(T, interval<T>);
template<class T> bool operator<=(interval<T>, interval<T>);
template<class T> bool operator<=(interval<T>, T);
template<class T> bool operator<=(T, interval<T>);
```

```
template<class T> bool operator>=(interval<T>, interval<T>);
template<class T> bool operator>=(interval<T>, T);
template<class T> bool operator>=(T, interval<T>);
}

// set inclusion comparison operators:
namespace set_inclusion_ops {
template<class T> bool operator==(interval<T>, interval<T>);
template<class T> bool operator==(interval<T>, T);
template<class T> bool operator==(T, interval<T>);
template<class T> bool operator!=(interval<T>, interval<T>);
template<class T> bool operator!=(interval<T>, T);
template<class T> bool operator!=(T, interval<T>);
template<class T> bool operator<(interval<T>, interval<T>);
template<class T> bool operator<(interval<T>, T);
template<class T> bool operator<(T, interval<T>);
template<class T> bool operator>(interval<T>, interval<T>);
template<class T> bool operator>(interval<T>, T);
template<class T> bool operator>(T, interval<T>);
template<class T> bool operator<=(interval<T>, interval<T>);
template<class T> bool operator<=(interval<T>, T);
template<class T> bool operator<=(T, interval<T>);
template<class T> bool operator>=(interval<T>, interval<T>);
template<class T> bool operator>=(interval<T>, T);
template<class T> bool operator>=(T, 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>&, interval<T>);

// value functions:
template<class T> T inf(interval<T>);
template<class T> T sup(interval<T>);
template<class T> T midpoint(interval<T>);
template<class T> T radius(interval<T>);

// set operations:
template<class T> bool is_empty_set(interval<T>);
template<class T> bool is_singleton(interval<T>);
template<class T> bool is_positively_bounded(interval<T>);
template<class T> bool is_negatively_bounded(interval<T>);
template<class T> bool is_bounded(interval<T>);
template<class T> bool contains(interval<T>, T);
template<class T> bool contains(interval<T>, interval<T>);
template<class T> bool equals(interval<T>, interval<T>);
template<class T> bool overlaps(interval<T>, interval<T>);
template<class T> bool comparable(interval<T>, interval<T>);
template<class T> interval<T> intersect(interval<T>, interval<T>);
template<class T> interval<T> hull(interval<T>, interval<T>);
template<class T> std::pair< interval<T>, interval<T> >
        split(interval<T>, T);
template<class T> std::pair< interval<T>, interval<T> >
        bisect(interval<T>);
```

```
// mathematical functions:
template<class T> interval<T> abs(interval<T>);
template<class T> interval<T> acos(interval<T>);
template<class T> interval<T> acosh(interval<T>);
template<class T> interval<T> asin(interval<T>);
template<class T> interval<T> asinh(interval<T>);
template<class T> interval<T> atan(interval<T>);
template<class T> interval<T> atanh(interval<T>);
template<class T> interval<T> atan2(interval<T>, interval<T>);
template<class T> std::pair< interval<T>, interval<T> >
        atan2_pair(interval<T>, interval<T>);
template<class T> interval<T> cbrt(interval<T>);
template<class T> interval<T> cos(interval<T>);
template<class T> interval<T> cosh(interval<T>);
template<class T> interval<T> erf(interval<T>);
template<class T> interval<T> erfc(interval<T>);
template<class T> interval<T> exp(interval<T>);
template<class T> interval<T> exp2(interval<T>);
template<class T> interval<T> expm1(interval<T>);
template<class T> interval<T> fabs(interval<T>);
template<class T> interval<T> fdim(interval<T>, interval<T>);
template<class T> interval<T> fma(interval<T>, interval<T>, interval<T>);
template<class T> interval<T> fmax(interval<T>, interval<T>);
template<class T> interval<T> fmin(interval<T>, interval<T>);
template<class T> interval<T> hypot(interval<T>, interval<T>);
template<class T> interval<T> ldexp(interval<T>, int);
template<class T> interval<T> lgamma(interval<T>);
template<class T> interval<T> log(interval<T>);
template<class T> interval<T> log10(interval<T>);
template<class T> interval<T> log1p(interval<T>);
template<class T> interval<T> log2(interval<T>);
template<class T> interval<T> nth_root(interval<T>, int);
template<class T> interval<T> pow(interval<T>, int);
template<class T> interval<T> pow(interval<T>, T);
template<class T> interval<T> pow(interval<T>, interval<T>);
template<class T> interval<T> pow(T, interval<T>);
template<class T> interval<T> sin(interval<T>);
template<class T> interval<T> sinh(interval<T>);
template<class T> interval<T> sqrt(interval<T>);
template<class T> interval<T> square(interval<T>);
template<class T> interval<T> tan(interval<T>);
template<class T> interval<T> tanh(interval<T>);
template<class T> interval<T> tgamma(interval<T>);

// mathematical partial functions:
template<class T> interval<T> acos(interval<T>, bool &);
template<class T> interval<T> acosh(interval<T>, bool &);
template<class T> interval<T> asin(interval<T>, bool &);
template<class T> interval<T> atanh(interval<T>, bool &);
template<class T> interval<T> divide(interval<T>, interval<T>, bool &);
template<class T> interval<T> divide(interval<T>, T, bool &);
template<class T> interval<T> divide(T, interval<T>, bool &);
template<class T> interval<T> lgamma(interval<T>, bool &);
template<class T> interval<T> log(interval<T>, bool &);
template<class T> interval<T> log10(interval<T>, bool &);
template<class T> interval<T> log1p(interval<T>, bool &);
template<class T> interval<T> log2(interval<T>, bool &);
```

```
template<class T> interval<T> nth_root(interval<T>, int, bool &);
template<class T> interval<T> pow(interval<T>, int, bool &);
template<class T> interval<T> pow(interval<T>, T, bool &);
template<class T> interval<T> pow(interval<T>, interval<T>, bool &);
template<class T> interval<T> pow(T, interval<T>, bool &);
template<class T> interval<T> sqrt(interval<T>, bool &);
template<class T> interval<T> tan(interval<T>, bool &);
template<class T> interval<T> tgamma(interval<T>, bool &);

// mathematical relations:
template<class T> interval<T> acos_rel(interval<T>, interval<T>);
template<class T> interval<T> acosh_rel(interval<T>, interval<T>);
template<class T> interval<T> asin_rel(interval<T>, interval<T>);
template<class T> interval<T> atan_rel(interval<T>, interval<T>);
template<class T> interval<T> atan2_rel(interval<T>, interval<T>);
template<class T> interval<T> div_rel(interval<T>, interval<T>);
template<class T> interval<T> nth_root_rel(interval<T>, int, interval<T>);
template<class T> interval<T> sqrt_rel(interval<T>, interval<T>);

} // of namespace std
```

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

```
namespace std {

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

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

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

  float lower() const;
  float upper() const;

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

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

  static interval<float> whole_set();
  static interval<float> empty_set();
```

```
};

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

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

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

  double lower() const;
  double upper() const;

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

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

  static interval<double> whole_set();
  static interval<double> empty_set();
};

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

  interval();
  interval(const char *);
  interval(long double t);
  interval(long double lo, long double hi);

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

  long double lower() const;
  long double upper() const;

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

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

  static interval<long double> whole_set();
  static interval<long double> empty_set();
};

} // of namespace std
```

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

1 **Note:** In the description of the member functions of a specialization `interval<T>` with `T` one of the three floating-point types `float`, `double`, and `long double`, the typename `U` designates a floating-point type different from `T`.

```
  interval();
```

2 **Effects:** Constructs an uninitialized interval.

```
  interval(const char *s);
```

3 **Effects:** Constructs an interval by extracting an interval from the NTBS pointed by `s`.

4 **Notes:** The interval is extracted with a `locale("C")` semantic. The behavior is undefined if the string pointed by `s` cannot be parsed as an interval.

```
  interval(T t);
```

5 **Effects:** Constructs an interval enclosing $\{\texttt{t}\}$.

6 **Postcondition:** `lower() == t && upper() == t` if `t` is a finite number.

7 **Notes:** Undefined if `t` is not a finite number.

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

8 **Effects:** Constructs an interval enclosing $\{x \mid \texttt{lo} \le x \le \texttt{hi}\}$.

9 **Notes:** Undefined if `lo` is neither a finite number nor $-\infty$, or if `hi` is neither a finite number nor $+\infty$, or if `lo` is not less or equal to `hi`.

```
  interval(U t);
```

10 **Effects:** Constructs an interval enclosing $\{\texttt{t}\}$.

11 **Notes:** Undefined if `t` is not a finite number.

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

12 **Effects:** Constructs an interval enclosing $\{x \mid \texttt{lo} \le x \le \texttt{hi}\}$.

13 **Notes:** Undefined if `lo` is neither a finite number nor $-\infty$, or if `hi` is neither a finite number nor $+\infty$ or if `lo` is not less or equal to `hi`.

```
  interval(interval<U> X);
```

14 **Effects:** Constructs an interval enclosing $\{x \in \texttt{X}\}$.

15 **Postcondition:** `is_empty_set(*this) == is_empty_set(X)`.

```
T lower() const;
```

16 **Returns:** a finite lower bound of `*this` if there is one, `-std::numeric_limits<T>::infinity()` otherwise.

17 **Notes:** Undefined if `*this` is empty. Implementation-defined if there is no finite lower bound and `std::numeric_limits<T>::has_infinity == false`.

```
T upper() const;
```

18 **Returns:** a finite upper bound of `*this` if there is one, `std::numeric_limits<T>::infinity()` otherwise.

19 **Notes:** Undefined if `*this` is empty. Implementation-defined if there is no finite upper bound and `std::numeric_limits<T>::has_infinity == false`.

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

```
interval& operator+=(T rhs);
```

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

```
interval& operator-=(T rhs);
```

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

```
interval& operator*=(T rhs);
```

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

```
interval& operator/=(T rhs);
```

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

```
interval& operator+=(interval rhs);
```

5 **Effects:** Stores an enclosure of $\{x+y \mid x \in$ `*this` and $y \in$ `rhs`$\}$ in `*this`.

6 **Returns:** `*this`.

```
interval& operator-=(interval rhs);
```

7 **Effects:** Stores an enclosure of $\{x-y \mid x \in$ `*this` and $y \in$ `rhs`$\}$ in `*this`.

8 **Returns:** `*this`.

```
interval& operator*=(interval rhs);
```

9 **Effects:** Stores an enclosure of $\{x \times y \mid x \in$ `*this` and $y \in$ `rhs`$\}$ in `*this`.

10 **Returns:** `*this`.

```
interval& operator/=(interval rhs);
```

11 **Effects:** Stores an enclosure of $\{x/y \mid x \in$ `*this` and $y \in$ `rhs` and $y \neq 0\}$ in `*this`.

12 **Returns:** `*this`.

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

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

1 **Notes:** Unary operator

2 **Returns:** `interval<T>(x)`

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

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

```
template<class T> interval<T> operator-(interval<T> X);
```

4 **Notes:** Unary operator.

5 **Returns:** a tight enclosure of $\{-x \mid x \in$ `X`$\}$.

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

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

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

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

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

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

26.6.6 `bool_set` comparisons [lib.interval.comps.bool_set]

1 The equality and relational comparison operators on intervals defined in namespace `bool_set_ops` are pure interval extensions of the corresponding comparison operators on the value type, using the inclusion property, and thus return an object of type `bool_set`. 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 `bool_set` when one or two arguments are empty, and `bool_set::indeterminate()` otherwise.

2 **Notes:** The arguments `lhs` and `rhs` of type `T` are implicitly converted to type `interval<T>`. The behavior of the comparison operators is undefined if they are not finite numbers.

```
template<class T> bool_set operator==(interval<T> lhs, interval<T> rhs);
template<class T> bool_set operator==(interval<T> lhs, T rhs);
template<class T> bool_set operator==(T lhs, interval<T> rhs);
```

3 **Returns:** an empty `bool_set` if either or both of `lhs` or `rhs` is empty, `false` if `overlaps(lhs, rhs)` is false, `true` if both contain the same single value, and `bool_set::indeterminate()` otherwise.

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

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

```
template<class T> bool_set operator<(interval<T> lhs, interval<T> rhs);
template<class T> bool_set operator<(interval<T> lhs, T rhs);
template<class T> bool_set operator<(T lhs, interval<T> rhs);
```

5 **Returns:** an empty `bool_set` if either or both of `lhs` or `rhs` is empty, `true` if every value in `lhs` is smaller than every value in `rhs`, `false` if every value in `lhs` is larger or equal to every value in `rhs`, `bool_set::indeterminate()` otherwise.

```
template<class T> bool_set operator<=(interval<T> lhs, interval<T> rhs);
template<class T> bool_set operator<=(interval<T> lhs, T rhs);
template<class T> bool_set operator<=(T lhs, interval<T> rhs);
```

6 **Returns:** `!( rhs < lhs )`.
7 **Notes:** `lhs <= rhs` returns the same `bool_set` as `lhs < rhs | lhs == rhs`.

```
template<class T> bool_set operator>(interval<T> lhs, interval<T> rhs);
template<class T> bool_set operator>(interval<T> lhs, T rhs);
template<class T> bool_set operator>(T lhs, interval<T> rhs);
```

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

```
template<class T> bool_set operator>=(interval<T> lhs, interval<T> rhs);
template<class T> bool_set operator>=(interval<T> lhs, T rhs);
template<class T> bool_set operator>=(T lhs, interval<T> rhs);
```

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

26.6.7 Possibly comparisons [lib.interval.comps.possibly]

1 The equality and relational comparison operators on intervals defined in namespace `possibly_ops` return `true` when there exists a pair of values taken in the arguments that satisfies the relation, `false` otherwise.

2 **Notes:** The arguments `lhs` and `rhs` of type `T` are implicitly converted to type `interval<T>`. The behavior of the comparison operators is undefined if they are not finite numbers.

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

3 **Returns:** `true` if there are two values $x \in$ `lhs` and $y \in$ `rhs` such that $x = y$, `false` otherwise.

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

4 **Returns:** `true` if there are two values $x \in$ `lhs` and $y \in$ `rhs` such that $x \neq y$, `false` otherwise.

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

5 **Returns:** `true` if there are two values $x \in$ `lhs` and $y \in$ `rhs` such that $x < y$, `false` otherwise.

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

6 **Returns:** `true` if there are two values $x \in$ `lhs` and $y \in$ `rhs` such that $x \leq y$, `false` otherwise.

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

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

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

8 **Returns:** `rhs <= lhs`.

26.6.8 Certainly comparisons [lib.interval.comps.certainly]

1 The equality and relational comparison operators on intervals defined in namespace `certainly_ops` return `true` when the relation holds true for any pair of values taken in the arguments, `false` otherwise.

2 **Notes:** The arguments `lhs` and `rhs` of type `T` are implicitly converted to type `interval<T>`. The behavior of the comparison operators is undefined if they are not finite numbers.

3 **Notes:** The result of the comparison operators is `true` if any input interval is empty.

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

4 **Returns:** `true` if $x = y$ holds true for any two values $x \in$ `lhs` and $y \in$ `rhs`, `false` otherwise.

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

5 **Returns:** `true` if $x \neq y$ holds true for any two values $x \in$ `lhs` and $y \in$ `rhs`, `false` otherwise.

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

6 **Returns:** `true` if $x < y$ holds true for any two values $x \in$ `lhs` and $y \in$ `rhs`, `false` otherwise.

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

7 **Returns:** `true` if $x \leq y$ holds true for any two values $x \in$ `lhs` and $y \in$ `rhs`, `false` otherwise.

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

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

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

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

26.6.9 Set inclusion comparisons [lib.interval.comps.inclusion]

1 The equality and relational comparison operators on intervals defined in namespace `set_inclusion_ops` return `true` when the relation holds true with respect to the set inclusion partial order, `false` otherwise.

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

2 **Returns:** `equals(interval<T>(lhs), interval<T>(rhs))`.

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

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

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

4 **Returns:** `contains(interval<T>(rhs), interval<T>(lhs)) && lhs != rhs`.

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

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

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

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

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

7 **Returns:** `rhs <= lhs`.

26.6.10 `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`, `[]`, `[t]`, 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.
5 **Notes:** The extraction respects the inclusion property: the interval *i* shall enclose all the values contained in the read interval.

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

6 **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, interval<T> i)
{
  basic_ostringstream<charT, traits> s;
  if (is_empty_set(i))
    s << "[]";
  else {
    s.flags(os.flags());
    s.imbue(os.getloc());
    s.precision(os.precision());
    s << "[" << i.lower() << ";" << i.upper() << "]";
  }
  return os << s.str();
}
```

7 **Returns:** `os`
8 **Notes:** The insertion respects the inclusion property: the written interval shall enclose all the values contained in the interval *i*.

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

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

1 **Returns:** `x.lower()` when `x` is not empty, `std::numeric_limits<T>::infinity()` otherwise.
2 **Notes:** Implementation-defined if `x` is empty and `std::numeric_limits<T>::has_infinity == false`.

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

3 **Returns:** `x.upper()` when `x` is not empty, `-std::numeric_limits<T>::infinity()` otherwise.
4 **Notes:** Implementation-defined if `x` is empty and `std::numeric_limits<T>::has_infinity == false`.

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

5 **Returns:** a finite number in `x` when `x` is not empty, and an implementation-defined value otherwise.
6 **Notes:** When `x` is a bounded interval, the result should approximate the real number $\frac{1}{2}(\texttt{inf(x)} + \texttt{sup(x)})$.

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

7 **Returns:** An implementation-defined value, neither positive nor 0, when `x` is empty, `T(0)` when `x` is a singleton, a finite number r such that interval $[\texttt{midpoint(x)} - r, \texttt{midpoint(x)} + r]$ encloses `x` when `x` is bounded, `std::numeric_limits<T>::infinity()` otherwise.
8 **Notes:** Implementation-defined if `x` is not bounded and `std::numeric_limits<T>::has_infinity == false`.

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

```
template<class T> bool is_empty_set(interval<T> x);
```

1 **Returns:** `true` if `x` is the empty set, `false` otherwise.

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

2 **Returns:** `true` if `x` contains a single real number and this value can be represented by a floating-point number of type `T`, `false` otherwise.

```
template<class T> bool is_positively_bounded(interval<T> x);
```

3 **Returns:** `true` if there exists a finite floating-point number of type `T` larger or equal to any value contained in `x`, `false` otherwise.

```
template<class T> bool is_negatively_bounded(interval<T> x);
```

4 **Returns:** `true` if there exists a finite floating-point number of type `T` smaller or equal to any value contained in `x`, `false` otherwise.

```
template<class T> bool is_bounded(interval<T> x);
```

5 **Returns:** `is_positively_bounded(x) && is_negatively_bounded(x)`.

```
template<class T> bool equals(interval<T> x, interval<T> y);
```

6 **Returns:** `true` if both `x` and `y` are empty, `true` if neither `x` nor `y` is empty and they contain the same values, `false` otherwise.

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

7 **Returns:** `true` if any real number contained in `rhs` is contained in `lhs`, `false` otherwise.

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

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

```
template<class T> bool overlaps(interval<T> x, interval<T> y);
```

9 **Returns:** `true` if the intersection `x`∩`y` is not empty, `false` otherwise.

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

10 **Returns:** `true` if neither `x` nor `y` is empty and the intersection `x`∩`y` is the empty set, `false` otherwise.

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

11 **Returns:** an empty `interval<T>` if `overlaps(x, y)` is false, an enclosure of the intersection `x`∩`y` otherwise.

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

12 **Returns:** `x` if `y` is empty, `y` if `x` is empty, an enclosure of the union `x`∪`y` otherwise.

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

13 **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`.

14 **Notes:** Returns a pair of two empty `interval<T>` if `x` is empty. Otherwise, the first member will be empty if and only if no value of `x` is smaller or equal to `t`, and the second member will be empty if and only if no value of `x` is larger or equal to `t`. Undefined if `t` is not a finite number.

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

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

26.6.13 `interval` mathematical functions [lib.interval.math.funcs]

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

1 **Returns:** an enclosure of $\{|x| \mid x \in \mathtt{X}\}$.

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

2 **Returns:** an enclosure of $\{\operatorname{acos}(x) \mid x \in \mathtt{X} \text{ and } x \in [-1,1]\}$.

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

3 **Returns:** an enclosure of $\{\operatorname{acosh}(x) \mid x \in \mathtt{X} \text{ and } x \geq 1\}$.

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

4 **Returns:** an enclosure of $\{\operatorname{asin}(x) \mid x \in \mathtt{X} \text{ and } x \in [-1,1]\}$.

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

5 **Returns:** an enclosure of $\{\operatorname{asinh}(x) \mid x \in \mathtt{X}\}$.

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

6 **Returns:** an enclosure of $\{\operatorname{atan}(x) \mid x \in \mathtt{X}\}$.

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

7 **Returns:** an enclosure of $\{\operatorname{atanh}(x) \mid x \in \mathtt{X} \text{ and } x \in (-1,1)\}$.

```
template<class T> interval<T> atan2(interval<T> Y, interval<T> X);
```

8 **Returns:** an enclosure of $\{\operatorname{atan2}(y,x) \mid x \in \mathtt{X} \text{ and } y \in \mathtt{Y}\}$.

```
template<class T> std::pair< interval<T>, interval<T> >
        atan2_pair(interval<T>, interval<T>);
```

9 **Returns:** a pair of intervals (Z_1, Z_2) such that $Z_1 \cup Z_2$ is an enclosure of $\{\operatorname{atan2}(y,x) \mid x \in \mathtt{X} \text{ and } y \in \mathtt{Y}\}$.
10 **Notes:** Z_1 and Z_2 do not overlap. When Z_2 is not empty, Z_1 is not.

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

11 **Returns:** an enclosure of $\{\sqrt[3]{x} \mid x \in \mathtt{X}\}$.

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

12 **Returns:** an enclosure of $\{\cos(x) \mid x \in \mathtt{X}\}$.

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

13 **Returns:** an enclosure of $\{\cosh(x) \mid x \in \mathtt{X}\}$.

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

14 **Returns:** an enclosure of $\{\operatorname{erf}(x) \mid x \in \mathtt{X}\}$.

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

15 **Returns:** an enclosure of $\{\mathrm{erfc}(x) \mid x \in \mathtt{X}\}$.

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

16 **Returns:** an enclosure of $\{\exp(x) \mid x \in \mathtt{X}\}$.

```
template<class T> interval<T> exp2(interval<T> X);
```

17 **Returns:** an enclosure of $\{2^x \mid x \in \mathtt{X}\}$.

```
template<class T> interval<T> expm1(interval<T> X);
```

18 **Returns:** an enclosure of $\{\exp(x) - 1 \mid x \in \mathtt{X}\}$.

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

19 **Returns:** an enclosure of $\{|x| \mid x \in \mathtt{X}\}$.

```
template<class T> interval<T> fdim(interval<T> X, interval<T> Y);
```

20 **Returns:** an enclosure of $\{\max(x-y, 0) \mid x \in \mathtt{X}$ and $y \in \mathtt{Y}\}$.

```
template<class T> interval<T>
        fma(interval<T> X, interval<T> Y, interval<T> Z);
```

21 **Returns:** an enclosure of $\{x \times y + z \mid x \in \mathtt{X}$ and $y \in \mathtt{Y}$ and $z \in \mathtt{Z}\}$.

```
template<class T> interval<T> fmax(interval<T> X, interval<T> Y);
```

22 **Returns:** an enclosure of $\{\max(x,y) \mid x \in \mathtt{X}$ and $y \in \mathtt{Y}\}$.

```
template<class T> interval<T> fmin(interval<T> X, interval<T> Y);
```

23 **Returns:** an enclosure of $\{\min(x,y) \mid x \in \mathtt{X}$ and $y \in \mathtt{Y}\}$.

```
template<class T> interval<T> hypot(interval<T> X, interval<T> Y);
```

24 **Returns:** an enclosure of $\{\sqrt{x^2+y^2} \mid x \in \mathtt{X}$ and $y \in \mathtt{Y}\}$.

```
template<class T> interval<T> ldexp(interval<T> X, int y);
```

25 **Returns:** an enclosure of $\{x \times 2^y \mid x \in \mathtt{X}\}$.

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

26 **Returns:** an enclosure of $\{\mathrm{lgamma}(x) \mid x \in \mathtt{X}$ and x not a non-positive integer$\}$.

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

27 **Returns:** an enclosure of $\{\log(x) \mid x \in \mathtt{X} \text{ and } x > 0\}$.

```
template<class T> interval<T> log10(interval<T> X);
```

28 **Returns:** an enclosure of $\{\log_{10}(x) \mid x \in \mathtt{X} \text{ and } x > 0\}$.

```
template<class T> interval<T> log1p(interval<T> X);
```

29 **Returns:** an enclosure of $\{\log(1+x) \mid x \in \mathtt{X} \text{ and } x > 0\}$.

```
template<class T> interval<T> log2(interval<T> X);
```

30 **Returns:** an enclosure of $\{\log_2(x) \mid x \in \mathtt{X} \text{ and } x > 0\}$.

```
template<class T> interval<T> nth_root(interval<T> X, int y);
```

31 **Returns:** an enclosure of $\{\sqrt[y]{x} \mid x \in \mathtt{X}\}$.
32 **Note:** Undefined if `y` is not positive.

```
template<class T> interval<T> pow(interval<T> X, int y);
```

33 **Returns:** an enclosure of $\{x^y \mid x \in \mathtt{X}\}$.

```
template<class T> interval<T> pow(interval<T> X, T y);
```

34 **Returns:** an enclosure of $\{x^y \mid x \in \mathtt{X}\}$.

```
template<class T> interval<T> pow(interval<T> X, interval<T> Y);
```

35 **Returns:** an enclosure of $\{x^y \mid x \in \mathtt{X} \text{ and } y \in \mathtt{Y}\}$.

```
template<class T> interval<T> pow(T x, interval<T> Y);
```

36 **Returns:** an enclosure of $\{x^y \mid y \in \mathtt{Y}\}$.

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

37 **Returns:** an enclosure of $\{\sin(x) \mid x \in \mathtt{X}\}$.

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

38 **Returns:** an enclosure of $\{\sinh(x) \mid x \in \mathtt{X}\}$.

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

39 **Returns:** an enclosure of $\{\sqrt{x} \mid x \in \mathtt{X} \text{ and } x > 0\}$.

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

40 **Returns:** an enclosure of $\{x^2 \mid x \in \mathtt{X}\}$.

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

41 **Returns:** an enclosure of $\{\tan(x) \mid x \in \mathtt{X} \text{ and } \cos(x) \neq 0\}$.

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

42 **Returns:** an enclosure of $\{\tanh(x) \mid x \in \mathtt{X}\}$.

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

43 **Returns:** an enclosure of $\{\text{tgamma}(x) \mid x \in \mathtt{X} \text{ and } x \text{ not a non-positive integer}\}$.

26.6.14 `interval` mathematical partial functions [lib.interval.math.part.funcs]

1 In addition to returning the same results as functions of 26.6.13, the following functions raise a flag passed as a parameter when their input intervals contain values outside the domain of the mathematical function on real numbers. They never clear the flag.

```
template<class T> interval<T> acos(interval<T> x, bool &f);
```

2 **Returns:** `acos(x)`.
3 **Postcondition:** If `f` was left unchanged to `false`, `x` did not contain values outside the interval $[-1,1]$.

```
template<class T> interval<T> acosh(interval<T> x, bool &f);
```

4 **Returns:** `acosh(x)`.
5 **Postcondition:** If `f` was left unchanged to `false`, `x` did not contain values less than 1.

```
template<class T> interval<T> asin(interval<T> x, bool &f);
```

6 **Returns:** `asin(x)`.
7 **Postcondition:** If `f` was left unchanged to `false`, `x` did not contain values outside the interval $[-1,1]$.

```
template<class T> interval<T> atanh(interval<T> x, bool &f);
```

8 **Returns:** `atanh(x)`.
9 **Postcondition:** If `f` was left unchanged to `false`, `x` did not contain values outside the interval $(-1,1)$.

```
template<class T> interval<T> divide(interval<T> x, interval<T> y, bool &f);
template<class T> interval<T> divide(interval<T> x, T y, bool &f);
template<class T> interval<T> divide(T x, interval<T> y, bool &f);
```

10 **Returns:** `x / y`.
11 **Postcondition:** If `f` was left unchanged to `false`, `y` did not contain zero.

```
template<class T> interval<T> lgamma(interval<T> x, bool &f);
```

12 **Returns:** `lgamma(x)`.
13 **Postcondition:** If `f` was left unchanged to `false`, `x` did not contain non-positive integers.

```
template<class T> interval<T> log(interval<T> x, bool &f);
```

14 **Returns:** `log(x)`.
15 **Postcondition:** If `f` was left unchanged to `false`, `x` did not contain non-positive values.

```
template<class T> interval<T> log10(interval<T> x, bool &f);
```

16 **Returns:** `log10(x)`.
17 **Postcondition:** If `f` was left unchanged to `false`, `x` did not contain non-positive values.

```
template<class T> interval<T> log1p(interval<T> x, bool &f);
```

18 **Returns:** `log1p(x)`.
19 **Postcondition:** If `f` was left unchanged to `false`, `x` did not contain non-positive values.

```
template<class T> interval<T> log2(interval<T> x, bool &f);
```

20 **Returns:** `log2(x)`.
21 **Postcondition:** If `f` was left unchanged to `false`, `x` did not contain non-positive values.

```
template<class T> interval<T> nth_root(interval<T> x, int y, bool &f);
```

22 **Returns:** `nth_rool(x, y)`.
23 **Postcondition:** If `f` was left unchanged to `false`, `x` did not contain negative values while `y` was an even integer.

```
template<class T> interval<T> pow(interval<T> x, int y, bool &f);
template<class T> interval<T> pow(interval<T> x, T y, bool &f);
template<class T> interval<T> pow(interval<T> x, interval<T> y, bool &f);
template<class T> interval<T> pow(T x, interval<T> y, bool &f);
```

24 **Returns:** `pow(x, y)`.
25 **Postcondition:** If `f` was left unchanged to `false`, neither `x` contained negative values while `y` was not a single integer, nor `x` contained zero while `y` contained a non-positive value.

```
template<class T> interval<T> sqrt(interval<T> x, bool &f);
```

26 **Returns:** `sqrt(x)`.
27 **Postcondition:** If `f` was left unchanged to `false`, `x` did not contain negative values.

```
template<class T> interval<T> tan(interval<T> x, bool &f);
```

28 **Returns:** `tan(x)`.
29 **Postcondition:** If `f` was left unchanged to `false`, `x` did not contain values of the form $k \cdot \frac{\pi}{2}$ with k an odd integer.

```
template<class T> interval<T> tgamma(interval<T> x, bool &f);
```

30 **Returns:** `tgamma(x)`.
31 **Postcondition:** If `f` was left unchanged to `false`, `x` did not contain non-positive integers.

26.6.15 `interval` mathematical relations [lib.interval.math.rels]

```
template<class T> interval<T> acos_rel(interval<T> X, interval<T> R);
```

1 **Returns:** an enclosure of $\{r \in \mathtt{R} \mid \cos(r) \in \mathtt{X}\}$.

```
template<class T> interval<T> acosh_rel(interval<T> X, interval<T> R);
```

2 **Returns:** an enclosure of $\{r \in \mathtt{R} \mid \cosh(r) \in \mathtt{X}\}$.

```
template<class T> interval<T> asin_rel(interval<T> X, interval<T> R);
```

3 **Returns:** an enclosure of $\{r \in \mathtt{R} \mid \sin(r) \in \mathtt{X}\}$.

```
template<class T> interval<T> atan_rel(interval<T> X, interval<T> R);
```

4 **Returns:** an enclosure of $\{r \in \mathtt{R} \mid \tan(r) \in \mathtt{X}\}$.

```
template<class T> interval<T>
        atan2_rel(interval<T> Y, interval<T> X, interval<T> R);
```

5 **Returns:** an enclosure of $\{r \in \mathtt{R} \mid \cos(r) \in \mathtt{X}$ and $\sin(r) \in \mathtt{Y}\}$.

```
template<class T> interval<T> div_rel(interval<T> X, interval<T> Y);
```

6 **Returns:** an enclosure of $\{r \in \mathbb{R} \mid \exists y \in \mathtt{Y},\ r \cdot y \in \mathtt{X}\}$.

```
template<class T> interval<T> nth_root_rel(interval<T> X, int y, interval<T> R);
```

7 **Returns:** an enclosure of $\{r \in \mathtt{R} \mid r^y \in \mathtt{X}\}$.
8 **Note:** Undefined if `y` is not positive.

```
template<class T> interval<T> sqrt_rel(interval<T> X, interval<T> R);
```

9 **Returns:** an enclosure of $\{r \in \mathtt{R} \mid r^2 \in \mathtt{X}\}$.

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

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

1 **Returns:** the interval containing all the real numbers.

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

2 **Returns:** the empty interval.

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 partial 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 radius of subintervals is <= precision.
template < class Function, class OutputIterator, class T >
OutputIterator
solve(Function f, OutputIterator oit, interval<T> current, T 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() );
  if (contains(eps, y) || radius(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 `radius` has been extended to vectors of interval (by taking max of radii) and that `assign_box(r,...)` assigns a small interval 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 radius of subintervals is <= precision.
template < class Function , class OutputIterator , class T >
OutputIterator
solve(Function f, OutputIterator oit,
      vector< interval<T> > const& current, T 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()));
  if (contains(r, res) || radius(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;-0.82287565513979643583]
[1.8228756554890424013;1.8228756560711190104]
                         [-0.82287565572187304497;-0.82287565513979643583]
[-0.8228756563039496541;-0.82287565572187304497]
                             [1.8228756549069657922;1.8228756554890424013]
[-0.82287565572187304497;-0.82287565513979643583]
                             [1.8228756549069657922;1.8228756554890424013]
[-0.82287565572187304497;-0.82287565513979643583]
                             [1.8228756554890424013;1.8228756560711190104]
```

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.

We also would like to thank a lot of people for their comments on the initial version of this proposal : Lawrence Crowl, Steven Clamage and their colleagues at Sun, George Corliss, John Pryce and Baker Kearfott from the Interval Subroutine Library project, and a lot of people having made comments on parts of the proposal : Stefan Schirra, Joris Van Der Hoeven, Jean-Pierre Merlet, Bernard Mourrain, Ron Avitzur, Lee Winter, Gabriel Dos Reis...

Finally, thanks to the Library Working Group for the comments and positive feedback in Mont-Tremblant, Berlin and Portland.

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://perso.ens-lyon.fr/guillaume.melquiond/doc/06-tcs-rnc5.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. *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://perso.ens-lyon.fr/nathalie.revol/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/app/docs/doc/819-3696`

[15] G. William Walster. *The Extended Real Interval System*. Manuscript, 1998.