

# A Proposal to Add Interval Arithmetic to the C++ Standard Library

Hervé Brönnimann    Guillaume Melquiond    Sylvain Pion    John Pryce \*

June 7, 2006

## Abstract

Proposed text to specify an interval class in the C++ standard library, with justification and background.

## Contents

<b>I</b>	<b>Motivation and Scope</b>	<b>3</b>
<b>II</b>	<b>Foundations</b>	<b>3</b>
<b>III</b>	<b>Impact on the Standard</b>	<b>5</b>
<b>IV</b>	<b>Design Decisions</b>	<b>5</b>
	IV.1 Design overview . . . . .	5
	IV.2 Abstract model issues . . . . .	5
	IV.3 Data representation . . . . .	7
	IV.4 API issues . . . . .	9
	IV.5 Relation to other parts of the standard library . . . . .	12
	IV.6 Other issues . . . . .	13
<b>V</b>	<b>Proposed Text for the Standard</b>	<b>15</b>
<b>26.6</b>	<b>Interval numbers [lib.interval.numbers]</b>	<b>15</b>
	26.6.1 Header <interval> synopsis [lib.interval.synopsis] . . . . .	16
	26.6.2 Interval class template [lib.interval] . . . . .	18
	26.6.3 Interval member functions [lib.interval.members] . . . . .	19
	26.6.4 Interval member operators [lib.interval.members.ops] . . . . .	20
	26.6.5 Interval non-member operations [lib.interval.ops] . . . . .	21
	26.6.6 Extended division function [lib.interval.xdiv] . . . . .	22
	26.6.7 <code>boolset</code> type and functions [lib.interval.boolset] . . . . .	22
	26.6.8 Interval comparison operators [lib.interval.compare] . . . . .	23
	26.6.9 Interval value operations [lib.interval.value.ops] . . . . .	23
	26.6.10 Interval mathematical functions [lib.interval.math] . . . . .	23
	26.6.11 Interval set operations [lib.interval.set.ops] . . . . .	25
	26.6.12 Interval static value operations [lib.interval.static.value.ops] . . . . .	26

---

\* *Disclaimer.* Though this document bears the names of Brönnimann, Melquiond and Pion, it is the result of work by John Pryce on their earlier version dated March 2006, and there has not yet been discussion on whether they accept the document in this revised form.

26.6.13 Interval I/O operations [lib.interval.io]	26
26.6.14 Interval I/O conversion [lib.interval.ioconversion]	27
26.6.A Design notes (informative)	27
26.6.A.1 Tables of interval cset values	27
26.6.A.2 Pseudo-code for the BAOs	27
26.6.A.3 The DISCONTINUOUS flag	27
26.6.A.4 Guidelines for unit testing	28
26.6.A.5 Test cases for unit tests of each function	28
26.6.A.6 etc...	28
<b>VI Examples of usage of the interval class.</b>	<b>29</b>
VI.1 Unidimensional solver	29
VI.2 Multi-dimensional solver	30
<b>VII Acknowledgements</b>	<b>32</b>

# 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. A mathematical interval  $[a, b]$  is the set of numbers lying between the two bounds  $a$  and  $b$ . The most important use is in computing proven inclusions of the range of a function  $f$ . Such an inclusion is a set containing all the values  $f$  takes as its arguments vary over given input intervals. R. Moore's **Fundamental Theorem** (1965), the basis of the theory, asserts that if the standard functions (including arithmetic operations) are given interval versions that possess this inclusion property, and if a function  $f$  is built from standard functions, then the corresponding interval version of  $f$  possesses the inclusion property, for any interval inputs that do not cause an exception to occur.

This standard is aimed at skilled programmers — applications developers and numerical analysts. This is because IA is harder to use effectively than it may first appear. Replace `double` by `interval<double>` in a code to solve simultaneous linear equations by Gaussian elimination, and you will obtain correct inclusions, but they will probably all be  $[-\infty, +\infty]$ . Do the same in a Runge-Kutta code for ordinary differential equations, and the interval output will certainly not be guaranteed to enclose the true solution. To solve nontrivial problems one needs different algorithms from those for point computation; those who ignore this only invite disappointment.

Naturally, we wish the widest possible usage of interval software, but we recommend implementers to place a prominent health warning on documentation, asking those inexperienced in numerical computation to learn about proper interval techniques and the pitfalls that await the unwary.

There exist many implementations of IA. See [2, 4, 12, 15, 19, 16] for six typical C++ implementations; more can be found on the web page [13], 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.

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

There are several kinds of usage of interval arithmetic [7, 13, 11]. The web page [13] gathering information about interval computations provides, among other things, a survey [9] of the subject and its application domains. We list a few here.

- Controlling rounding errors of floating point computations at run time.
- Solving systems of linear, nonlinear 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]).

## II Foundations

*What is the theoretical underpinning of the proposal? What choices are there at the theory level?*

An interval is just a subset of the set of numbers, one that is convenient to represent and compute with. There are IA systems, e.g. Kaucher arithmetic, for which intervals have structure beyond being just sets, but this proposal does not consider them. Thus there is little dispute about what “interval version of a function  $f$ ” means, provided the inputs to  $f$  are *bounded closed nonempty intervals*  $X = [\underline{x}, \bar{x}] = \{x \mid \underline{x} \leq x \leq \bar{x}\}$  in the real line  $\mathbb{R}$ , on which the function is everywhere defined and *continuous*. The function result is then defined to be the exact set-theory range, which by results of elementary real analysis is also a bounded closed nonempty interval. For an arithmetic operation  $\bullet$ , one of  $+ - \times \div$ , this means that  $X \bullet Y$  is given by

$$X \bullet Y = \{x \bullet y \mid x \in X \text{ and } y \in Y\}, \quad (1)$$

provided one excludes the case of division by an interval containing zero. What will be termed the *Simple System* supports only such intervals and raises an error if one tries, e.g., to construct  $[1, +\infty]$  or to compute  $1/[0, 1]$  or  $\sqrt{[-1, 1]}$ .

The main defect of the Simple System is that it cannot handle infinity. This is the more glaring since the widespread adoption of IEEE 754 arithmetic, with its mathematically well founded way to handle infinities. We see it as imperative for this Standard *to support infinity, in the context of IA, in a clearly defined and mathematically consistent way.*

Even with the constraint “intervals as just sets”, a great variety of IA systems exist in the literature. There are intervals in the complex plane defined as disks  $\{z \mid |z - c| \leq r\}$ , see [20]; or intervals in the rationals or other ordered fields.

However, this proposal is concerned only with IA systems based on the reals  $\mathbb{R}$ , since this is of most use in science and engineering. This still leaves many versions of IA that handle infinity, see [6, 8, 17, 21]. Some key differences between them are shown by the following questions.

1. Is  $\infty$  an actual number? For instance, is  $+\infty$  a member of the interval  $[1, +\infty]$ , or is it just a placeholder to indicate the upper bound?
2. Is there just one  $\infty$ , or signed  $-\infty$  and  $+\infty$ ?
3. Does the system support open and half-open intervals, that is intervals that omit one or both of their end points?
4. Does the system support the empty interval?
5. Does the system support *wrap-around* intervals  $[a, b] := \{x \mid x \geq a\} \cup \{x \mid x \leq b\}$  where  $a > b$ ?
6. Are there separate  $-0$  and  $+0$  that behave differently — not just by default at implementation-level, but as part of the theory?
7. How does the system handle standard functions where they are singular, such as  $\tan(x)$  at  $x = \pi/2$ , or discontinuous, such as  $\text{sign}(x)$  at  $x = 0$ ? Is the interval result defined purely in set theoretic terms by the formula (1), or are notions of continuity and limits involved?
8. How does the system handle points outside a function’s domain, as when evaluating  $\sqrt{[-1, 1]}$ ?

Useful diagnostic questions about a system are how it evaluates the following, where  $[a]$  denotes the singleton interval  $[a, a]$ .

$$[1]/[0], \quad [1]/[0, 1], \quad [1]/[-1, 1], \quad [0]/[0], \quad \log([-1, 1]).$$

This discussion shows that a well founded IA implementation requires a clear *abstract (mathematical) model* that defines, on an “exact arithmetic” machine:

- the underlying number system;
- what is meant by an interval, and the set of all allowed intervals in the IA system;
- the result of the interval version of any operation or function on given interval inputs, including at special points such as singularities.

Depending on how one answers the questions listed above, one may construct scores, maybe hundreds, of distinct models, any of which could be the basis for a workable **interval** type. This fact is often obscured, especially in documentation at the level of implementation.

We gave serious consideration to a standard that supports several abstract models, but chose a *one-model standard* as being simpler and much easier to make precise.

The model we propose is based on containment set (cset) theory, as is already used in the interval facility of the Sun C++ and Fortran compilers and as an option in the *filib++* package.

Cset theory is described in Pryce and Corliss [18], and Walster and Hansen [5]. Brönnimann, Melquiond and Pion [1] discuss IA in the context of a general ordered field, as well as interval representations, rounding modes, basic operations, and possible interval comparison operators.

### 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, using rounding mode change functions, which are available in `<cfenv>`. However, an efficient implementation of the proposal will rely on specific optimizations from the compiler (e.g., optimizing away redundant FPU rounding mode changes), which are outlined below.

### IV 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?*

#### IV.1 Design overview

The standard introduces a single template class `interval<T>`, where `<T>` is a real type. Operations on this type guarantee the inclusion property in the containment set sense, even in exceptional cases. That is, if a computation in the mathematical domain guarantees to enclose a certain set, so must its implemented version (the “Thou shalt not lie” principle). As with `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. The behavior on out-of-domain argument values is a silent and no-exception behavior, which returns an enclosure of the cset values at points in the function's domain and on the domain's boundary and ignores other points. For instance,  $\sqrt{[-1, 4]}$  returns  $[0, 2]$ . An implementation may raise a `DISCONTINUOUS` flag when this occurs.

#### IV.2 Abstract model issues

##### What is the model?

The chosen model is as follows.

- The whole line is the extended reals  $\mathbb{R}^* = \mathbb{R} \cup \{-\infty, +\infty\}$ .
- The set  $I$  of all supported intervals comprises the empty set, together with all nonempty closed intervals in  $\mathbb{R}^*$ , that is  $[a, b]$  for arbitrary  $a, b \in \mathbb{R}^*$  with  $a \leq b$ . This includes the whole line, since it equals  $[-\infty, +\infty]$ .
- Arithmetic operations and standard functions are defined in the cset sense. For points  $x$  and  $y \in \mathbb{R}^*$  and an arithmetic operation  $\bullet$ , the cset value of  $x \bullet y$  is the set of all possible limits of  $z_r = x_r \bullet y_r$ , where  $x_r \rightarrow x$  and  $y_r \rightarrow y$  as  $r \rightarrow \infty$ , and  $x_r \bullet y_r$  is defined in the ordinary real sense for all  $r$ . For subsets  $X$  and  $Y$  of  $\mathbb{R}^*$ , the cset value of  $X \bullet Y$  is the union of the cset values of  $x \bullet y$  over all  $x \in X$  and  $y \in Y$ .

When  $X$  and  $Y$  are intervals in  $\mathbb{R}^*$ , the mathematical result  $X \bullet Y$  (synonym: the *interval cset value*) is the smallest member of  $I$  containing the cset value of  $X \bullet Y$ .

The same notions for standard functions  $e(x, \dots)$  are defined analogously.

## Why just one model, and why csets?

The most established interval packages are based on the Simple Model. Their models generally lack a clear definition of how to handle infinite intervals, and/or their implementations do so inconsistently. It was attractive to create a standard to which these widely used packages would conform by minor modifications. This would encourage, for instance, standardizing of names for basic functions such as an interval's bounds (should the lower bound be called `lower`, `inf` or `lo`?) and so on. It would also require from implementations a more careful definition of how they handle infinity.

However, we wanted to allow a cset model and accordingly drafted a text that supported both.

Unfortunately, to state the various alternatives consistently turned out just too cumbersome. We therefore chose a one-model standard, and settled on a cset-based model. Some reasons for this choice are listed here.

1. Cset theory, though recent, is part of classical real analysis as it has existed for over a century.
2. The cset of *any* function  $f$  is unambiguously defined by  $f$ 's definition as a function of real variables (and by the extended number system chosen). This gives a completely uniform and consistent way to handle points where functions are infinite or discontinuous.
3. The cset version of Moore's Fundamental Theorem says that if the standard functions are given interval versions that enclose the exact cset, and if a function  $f$  is built from standard functions, then the corresponding interval cset version of  $f$  possesses the inclusion property for *all* interval inputs — evaluating a function in the cset sense never raises an exception.
4. Cset-based systems give the same result as traditional interval systems on inputs that the latter process successfully. This holds in exact arithmetic: in machine arithmetic it holds “up to roundoff” to an extent that depends on the implementation.
5. Csets allow compiler optimizations that other IA systems do not. This is a consequence of the Rational Rearrangement Theorem, which states that algebraically equivalent rational expressions have the same cset.
6. At least two C++ implementations of csets already exist.

Even within cset models, there are various possibilities. A model is uniquely defined by: the original number system (the real field  $\mathbb{R}$  is the obvious choice, but a model can equally be based on the complex field); the chosen extended number system (we chose the traditional extended reals  $\mathbb{R}^*$  as being most useful and compatible with IEEE arithmetic); and the definition of interval hull of a set (e.g., we could have supported wrap-around intervals, but decided against them).

## Division by an interval containing zero

There has been much disagreement on this in the literature. There are three common versions of division on sets,  $X/Y$ :

- (i) the set extension of point-division is  $\{z \mid x \in X, y \in Y, z = x/y\}$ ;
- (ii) (the set extension of) an inverse of multiplication is  $\{z \mid x \in X, y \in Y, x = y \times z\}$ ;
- (iii) the cset model is the set of all limits  $z_r = x_r/y_r$ , where the  $x_r$  converge to some point in  $X$ , the  $y_r$  converge to some point in  $Y$ , and  $y_r \neq 0$ .

The results only differ when  $0 \in Y$ . The following examples illustrate.

- (i) In a model based on the reals and with version (i),  $0/0$  is empty, and  $1/[0, 0.5]$  is  $[2, +\infty)$ . (A round bracket denotes an open end to an interval.)
- (ii) In a model based on the reals and with version (ii),  $0/0$  is  $(-\infty, +\infty)$ , and  $1/[0, 0.5]$  is  $[2, +\infty)$ .

- (iii) In the cset model of this standard, the exact cset of  $0/0$  (more precisely: of the function  $x/y$  at  $x = y = 0$ ) is  $[-\infty, +\infty]$ . The exact cset of  $1/[0, 0.5]$  is the union of  $[2, +\infty]$  and the singleton (that is, one-point) interval  $[-\infty]$ .

This becomes  $[-\infty, +\infty]$  after taking the interval hull: not a defect of csets but a property of the chosen interval hull operation. The extended division function `xdiv`, see §26.6.6 of the standard text, can be used to return the two interval parts separately in such cases.

### Operations on empty intervals

These must be consistent with the basic definition of intervals as sets. If  $f(x, y, \dots)$  is a point-function (arithmetic operation or standard function) and  $X, Y, \dots$  are sets, the “set version”  $f(X, Y, \dots)$  is empty if any of  $X, Y, \dots$  is empty. Hence, so is the cset and its interval hull. Of course, this does not apply to operations not derived from point functions, such as `hull(X, Y)`, which returns the interval hull of the set union  $X \cup Y$ .

### Wrap-around intervals?

A scheme first introduced by W. Kahan [8] is to support *wrap-around* intervals

$$[a, b] := \{x \mid x \geq a\} \cup \{x \mid x \leq b\} \quad \text{where } a > b,$$

which arise as a result of division by an interval containing zero. With the lower/upper bound representation, they can be stored as objects with `lower>upper`; with the midpoint/radius representation they can be stored as objects with `radius<0`. The exact cset result of any of  $+ - \times \div$  with normal or wrap-around interval arguments is again a normal or wrap-around interval. We think however that on typical current machine architectures, the tighter inclusion obtained for division by zero does not warrant the overhead introduced into the other arithmetic operations. Instead we provide the *extended division* operation `xdiv` to handle this case.

Hardware support could remove the performance penalty of wrap-around intervals.

### Behavior on out-of-range argument values?

Some functions of real variables are “partial”, meaning that their domain of definition  $\mathcal{D}_f$  is not the whole of their domain space. Examples are  $\sqrt{x}$ ,  $\log(x)$ , which are not defined on all of  $\mathbb{R}$ ; and the division function, which is not defined on all of  $\mathbb{R}^2$ . Csets implement the standard set-theory meaning — termed *loose evaluation* — of the range of such a function. Namely, if  $X$  is a subset of the domain space, then  $f(X)$  is the set of values  $f(x)$  over those  $x \in X$  for which  $f$  is defined.

For instance, this allows negative values in an argument to `sqrt` without throwing an exception. This permits silent error propagation and recovery for `sqrt(X)`, where  $X$  is (in theory) guaranteed to represent a positive number, but its computed enclosure contains negative numbers because of the inevitable overestimations of interval computation.

Because the cset is defined by limits, there is a little more to it. For instance  $\log$  is not defined at 0, but the cset value of  $\log$  at 0 is  $-\infty$ . In general, there is always a nonempty cset value at points that are outside  $\mathcal{D}_f$  but on its boundary: e.g.,  $\exp(-\infty) = 0$ ,  $\exp(+\infty) = +\infty$ ,  $\sin(+\infty) = [-1, 1]$  in the cset sense.

See the discussion of the `DISCONTINUOUS` flag in the Appendix, §26.6.A.3.

## IV.3 Data representation

### Representation of intervals

Two natural representations are:

- “lower/upper bound”, where  $[a, b]$  is represented by two numbers `{lower, upper}` (using braces to delimit a list of values), holding  $a$  and  $b$  respectively;
- “midpoint/radius”, where  $[a, b]$  is represented by `{mid, rad}` holding  $m$  and  $r$  such the interval is  $[m - r, m + r]$ .

Each has advantages, but this proposal generally discusses the bounds representation. Roundoff may occur when computing the midpoint and radius from the bounds, or vice versa. A variant is to store, say, `{mlower, upper}` holding  $-a$  and  $b$ , which makes it simpler to do all arithmetic operations using upward rounding only.

### Empty and infinite intervals

On IEEE machines and using the bounds representation, there is no problem with representing semi-infinite intervals and the whole line. One can represent the empty set by setting lower and upper to NaN. This lets the empty set propagate through the four arithmetic operations. An additional cost is caused by the need to test for cases such as  $[-\infty, 3] + [+ \infty, +\infty]$ , which generates a NaN in the lower bound but whose correct cset result is  $[-\infty, +\infty]$ . If this test is done by an explicit “if” in a high level language, the cost may be high on typical workstation architectures, because it flushes the pipeline. However, work by Branimir Lambov [14] shows that on a large number of such machines the instruction set lets one produce the same effect without flushing the pipeline. Thus the additional cost can be made negligible.

On machines that do not support infinity and NaN, it is possible to support infinite intervals and the empty set by using reverse ordered bounds, that is data with `lower > upper`. A possible scheme is as follows.

Let  $M$  be the largest finite machine number. Data `{M, b}` represents  $[-\infty, b]$  if  $-M < b < M$ . Data `{a, -M}` represents  $[a, +\infty]$  if  $-M < a < M$ . This covers all intervals whose endpoints are machine numbers or infinite, except the empty set, the whole line, and (should it be deemed necessary to support them) a few intervals such as  $[\pm M, \infty]$ . These may be represented, say, by `{1, 0}`, `{2, 0}`, ...

A single test per interval argument to a function separates these “unusual” cases from the “usual” case of finite, non-empty intervals. Thus, assuming the latter happen almost all the time, there is little loss of efficiency.

### Memory layout

We are aware of LWG issue 387 on over-encapsulation of `std::complex`. We have not required an interval to be represented by two numbers of type `T`, and instead simply require that an interval object occupy a contiguous block of memory, of size depending only on `T`. This supports efficient use of `valarray`, while not forbidding extra sophistications such as “tags” on intervals.

### Passing by value or by reference?

Since intervals are small objects, some application binary interfaces (ABIs) implement passing them by value more efficiently (in registers), but other ABIs prefer passing by reference. We have chosen to follow what was already in the standard for `std::complex`, which is passing by reference. Changing this to pass by value would only penalize the other ABIs. One option could be to make this dependent on the implementation, and communicate this choice to the user, for example by a nested type provided by interval.

This point is moot when functions are inline, which is expected to be the case most of the time. *[JDP Note: This item, and the function signatures in pass-by-reference form in §26.6.1, are retained from a previous draft. However ISL are convinced by the arguments of Lawrence Crowl and others in favour of pass-by-value. ]*

## IV.4 API issues

### What intervals should `interval<T>` support?

Let  $I_T$  denote the set of all intervals supported by the `interval<T>` instantiation. It is not required that  $I_T$  comprise all intervals with arbitrary endpoints of type `T` (even with the bounds representation). Implementers may disallow some of these to improve speed. For instance the system described by Jaulin et al. [7] supports all such intervals except for the singletons  $[-\infty, -\infty]$  and  $[\infty, \infty]$ ; the claimed benefit is to remove a test in the addition and subtraction operations.

### Integrity

Intervals are intended for high-reliability and safety-critical computations. Nothing can stop a C programmer writing code that oversteps array bounds or otherwise corrupts memory. However we consider preserving data encapsulation an important part of ensuring a reliable infrastructure. Therefore, the standard requires implementations to specify the set  $I_T$  (see above), and requires that it be impossible for methods of the `interval` class to create an `interval<T>` object that does not represent a member of  $I_T$ .

### Tightness of results

We do not require that the tightest intervals preserving the inclusion property be returned by arithmetic operations, since this can be hard to implement, or slow, on systems without IEEE 754 support. It is only necessary to preserve the containment (enclosure) property for all functions. Tightness is then a Quality Of Implementation (QOI) issue. There is leeway for implementers to trade it for speed.

### Exceptional cases for constructors

What should a constructor do when given nonsense arguments: for example,

```
interval(NaN, 1)?
```

What it must *never* do is construct an `interval` object that does not represent a valid interval. There are at least three natural choices: return `whole`; return `empty`; or raise an exception. Returning `whole` means “we know absolutely nothing about this number”. By contrast `empty` means in a sense that we know everything about a number, namely “we have proved there is no number satisfying the conditions of this calculation”. In the example above, the NaN may have arisen from:

```
double a,b,c;
a = some expression;
b = some expression;
c = a/b;
X = interval(c, 1);
```

If `a` and `b` happen to be zero, `c` equals NaN. If, in the underlying mathematics, division has its traditional meaning, then an appropriate interpretation is “There is no solution to this problem”, and `empty` is a suitable value for `X`. If division should be seen as the inverse of multiplication, an appropriate interpretation is “`c` can have any real value”, and `whole` (or more tightly,  $[-\infty, 1]$ ) is a suitable value for `X`. To avoid “lying”, `whole` is never an *incorrect* choice, but the best choice requires information only available at a level higher than the constructor.

Raising an exception is a good alternative. A case such as the above example probably arises from a coding error during development: a runtime exception will help pinpoint it. A counterargument is that efficient code is generally vectorized and, when such a constructor is called for each element of a long vector, an exception would flush the pipeline and impact performance. Against this, initializing

objects is usually a very small part of a large numerical computation. Also, constructors are critical points of the code where one should check *beforehand* that a valid object is being constructed; hence one should program in a style that prevents such exceptions occurring in production code.

On balance, we decided *interval constructors shall raise an exception* when given invalid arguments, that is when one or both arguments is NaN.

### Exceptional cases for point functions of intervals

A contrasting case is various point-valued functions of intervals — `midpoint`, `width`, etc. — that have an obvious value for nonempty bounded intervals, but no natural value for some cases beyond this. Examples are the midpoint of  $[0, +\infty]$  or the bounds and width of the empty set  $\emptyset$ .

Tradition based on mathematical consistency often offers a value: e.g.,  $\inf(\emptyset) = +\infty$ ,  $\sup(\emptyset) = -\infty$  make sense in the context of subsets of  $\mathbb{R}^*$ , which implies  $\text{width}(\emptyset) = -\infty - +\infty = -\infty$ . Failing this there are two natural design choices: return NaN, or raise an exception. We felt the presumption here is in favour of “silent, no-exceptions behaviour”. Hence functions consistently return NaN (on systems that support it) in such cases.

### What exceptions?

Decisions on whether existing exceptions should be used, or new ones created for the `interval` class, are deferred till this proposal has been discussed further.

### What types as template parameter?

The built-in floating point types `float`, `double` and `long double` must be supported. It is not much harder for an implementer to support all three than to support only one of them.

The standard simply requires enclosure of the exact result of an operation, and says nothing about how the enclosure is produced. Whether this is by outward rounding to the nearest representable interval, or some other way, is an implementation issue.

Hence *any* floating point type can be used as template parameter. This includes user-defined types, for instance arbitrary-precision or rational arithmetic. The decimal floating point types under discussion for inclusion in the IEEE 754 standard could also be used. They would have the advantage of making I/O conversion trivial.

Integral types as template parameter are incompatible with this proposal.

### Bounds functions

The commonest implementations will be on IEEE arithmetic systems using some version of the lower/upper bound data representation. However, so as not to penalize non-IEEE implementations, we have provided `lower()` and `upper()` that give access to the bounds, giving an implementation-defined value on the empty set but without the overhead of testing for that case; and also `inf()` and `sup()` that give the mathematically conventional value on the empty set at the cost of an extra test.

### Comparison operators

There is no comparison on intervals that creates a total order like that on the base type while also being useful in typical interval algorithms. Various numeric comparisons on intervals have been proposed, such as “possibly” and “certainly” operators. For example,  $X$  is possibly [resp. certainly]  $\leq Y$  if *for some* [resp. *for all*]  $x \in X$  and  $y \in Y$  one has  $x \leq y$ . All are potentially useful, but none massively so. If one implements them individually (as in the Sun interval system) there is quite a large number of them. We offer a cleaner scheme.

The mathematical essence is that *any* function that gives point output from point input extends to a function that gives set output from set input. The comparison operators map a pair of numbers to a boolean, so any such operator has an extension such as

$$X \leq Y = \{ x \leq y \mid x \in X \text{ and } y \in Y \},$$

that maps two sets of numbers (in particular intervals) to a set of booleans — that is, to one of `{false, true}`, `{true}`, `{false}`, or empty set  $\emptyset$ . For example `[0, 1] < [1, 2]` returns `{false, true}`, `[0, 1] ≤ [1, 2]` returns `{true}`. If  $\bullet$  is any of the six numeric comparison operators `=`, `≠`, `<`, `≤`, `>`, or `≥`, then `X • Y` is empty if, and only if, at least one of `X` and `Y` is empty. (This is because these operators are everywhere defined: `x • y` returns `true` or `false` for any numbers `x` and `y`.)

Adapting from a previous draft, the standard offers a type `boolset` representing “set of boolean”, and `boolset`-valued versions of the six comparisons.

`boolset` can be implemented in several ways. We make it a specialization of the bit-pattern class `bitset` in the Standard Library, a non-mutable bit-pattern of length 2:

```
typedef const std::bitset<2> boolset;
```

The advantage of this is that it completely decouples the “possibly/certainly” concept from the comparisons. We just provide three `bool`-valued functions of `boolset` called `possibly`, `certainly`, and `definitely` (the last being a stronger version of `certainly`). See the examples in §26.6.7 of the standard text.

### Why no setter versions of the getters `lower()` and `upper()`?

Code that can modify a bound of an `interval` object destroys encapsulation and makes correctness harder to prove. A bound-setter function has the overhead of preserving the invariant `lower ≤ upper`, and is complicated to implement with the midpoint/radius representation. Therefore we decided such a feature is not really useful.

### Supporting correct interval computation in the large

For a user to perform guaranteed interval computations, there is far more required than simply that every arithmetic operation must satisfy the inclusion property.

First, the right algorithms must be used — see the remarks on solving an ordinary differential equation (ODE), etc., in §I. Second, there must be clear thinking about when to enter and leave the “interval world”. For instance if a program inputs, as a point-value,  $y_{\text{init}} = 0.1$  as initial value for an ODE, then subsequent use of an interval ODE code can at best guarantee to enclose the solution corresponding to the stored value of 0.1, which is not 0.1 exactly in binary floating point. The current standard cannot help with these two issues.

Third, however, one can provide interval I/O that guarantees inclusion. An interval input function can read the string `[0.1]`, or just `0.1`, and convert it to a stored interval guaranteed to contain 0.1. Interval output should guarantee the same in reverse, on converting internal intervals to external, textual form. We decided such a facility must be in the standard.

The fourth issue extends the third and impacts the second. Conversion between external and internal form happens not only at run time with I/O, but also at compile time with literal constants in code. When executing, say, the code `x = interval<T>(0.1)`; under this standard, the 0.1 will be converted to floating point and then to a singleton interval, which will not contain the exact value 0.1. This is why we provide the alternative `x = interval<T>("0.1")`; which guarantees inclusion. To make the code `x = interval<T>(decstr)`; produce an enclosure of the exact number represented by a (not in quotes!) decimal literal `decstr`, one must change the language semantics and hence the compiler; one cannot do it by implementing intervals purely as a library.

That one language change would avoid many errors committed by inexperienced interval programmers, but no amount of such changes can ever absolve programmers from deciding what they really intend. For instance the declaration `double c = 0.1;` could be followed by `X = interval<T>(c);` later in the code. Then `X` does not contain 0.1. Was that the programmer’s intention? There is no way a compiler can tell.

Therefore we make no apologies that this standard does not offer the extra support for intervals that would require changing language semantics.

### Why the notation `[x; y]` for interval I/O?

Complex numbers already use the notation `(x,y)`, and the use of brackets is customary for representing an interval. We use the semicolon `;` instead of the comma `,` as the separator to avoid conflicts with locales (we believe none use `;` as a decimal point, but some do use the comma `,`).

### Changes from the original Brönnimann *et al.* proposal

Discussions with C++ experts indicated that the section giving “numeric specializations” for the three basic types `float`, `double` and `long double` was not necessary to the standard as now written, so it has been removed.

The API was discussed with workers, notably Kearfott and Nedialkov, who have extensive experience of interval computation in the areas of linear algebra, global optimization and differential equations. As a result the following changes were made:

- “Magnitude” and “mignitude” functions (the upper and lower bounds of  $|x|$  for  $x$  in an interval) were added.
- `contains(X,Y)` was replaced by `subsetq(Y,X)` on the ground that accepted mathematical style uses  $Y \subseteq X$  far more often than  $X \supseteq Y$ .
- For a similar reason, `overlap(X,Y)` — which tests for  $X \cap Y \neq \emptyset$  — was replaced by `disjoint(X,Y)` — which tests for  $X \cap Y = \emptyset$ .
- The boolean function `interior(X,Y)` was considered useful and added. This checks if interval  $X$  is contained in the interior of interval  $Y$ .
- The `is_singleton(X)` function was seen as unnecessary; one equivalent test is `width(X)==0`. The `comparable(X, Y)` function is just one of various comparisons that can be constructed from the `boolset` comparison scheme, and was removed.

## IV.5 Relation to other parts of the standard library

### Specializations for `numeric_limits<interval>` ?

We do not see a meaningful specialization.

### Should `std::valarray<interval>` work?

We believe this standard permits it. Its use is to be encouraged as a way to produce readable and efficient code at higher levels such as interval linear algebra.

### Should `std::set<interval>` work?

*See Library Issue 388 concerning `std::set<std::complex>`.*

The answer seems to be: Yes, at programmer’s risk. In order to safely use `std::set<interval>`, users have to provide a comparison operator that is guaranteed to implement a *strict weak order*, see C++ Standard, 25.3, on any set of intervals to be used. It could implement lexicographic order on the bounds, for example, but this does not have a relation to any natural ordering of intervals as sets of numbers.

If one can assert that the intervals in the set are pairwise disjoint, then the comparison “X is strictly to the left of Y” can be used, but it will throw an exception if the assertion fails. It appears the standard does not specify the behavior of `std::set` when a comparison throws an exception (even if that is caught right outside the call to `set::insert()` — it could for instance cause a memory leak).

## IV.6 Other issues

### Comparison with existing libraries

Rather than just putting together a common subset of features in existing C++ interval arithmetic libraries, we tried to propose a consistent and adequate set of features. The `boost.interval` library, which acted as a sandbox for this proposal, provides these features and many more (such as whether to support empty intervals, rounding methods, the meaning of comparisons, intervals over user-defined types, etc.) via a policy-based design. The current proposal is far less complex for the implementer and the user, which makes it easier to achieve the ultimate goal of high-reliability computation in science and engineering.

We considered the C++ libraries PROFIL, `filib++`, Gaol, and the 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 invisibly in each interval operation, or you set it globally and require users to be very careful with their 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. In Sun and in this proposal, rounding mode switches are invisible: 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).

Comparison operators between sets of numbers, in particular between intervals, include “certainly” and “possibly” comparisons and others. Boost provides these through namespace selection, the other libraries provide them through explicitly-named functions.

The Fortran community has also been very active in the domain of interval arithmetic, and this proposal is comparable with a proposal [10] that was written for this language.

### Relation to existing standards : IEEE-754 and LIA-123?

Run time choices of rounding modes are not part of the LIA standard, but it is possible to implement `interval` using only LIA features.

### Optimization expectations

One goal of the standardization of interval arithmetic is to make an implementation close to compilers, hence motivate optimization work. These are mostly QOI issues.

On most current architectures, rounding mode changes seriously impact pipeline speed. Efficiency is important for basic interval arithmetic operations. We mention two optimizations that can eliminate most rounding mode changes:

First, the addition  $a + b$  rounded towards  $-\infty$  is the same as  $-(-a - b)$  with operations rounded towards  $+\infty$ , so 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 for machines where double rounding can have an effect (e.g. x86).

Second, the compiler, provided it has knowledge of rounding mode change functions, can eliminate a rounding mode change if the new mode is the same as the current one. Coupled with the previous trick, this lets one eliminate rounding mode changes from a sequence of interval operations that has no floating point operations interspersed.

## 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 <code>&lt;complex&gt;</code>
26.3	Numeric arrays <code>&lt;valarray&gt;</code>
26.4	Generalized numeric operations <code>&lt;numeric&gt;</code>
26.5	C library <code>&lt;cmath&gt;</code> <code>&lt;cstdlib&gt;</code>
26.6	Interval arithmetic <code>&lt;interval&gt;</code>

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 functions for representing and manipulating numerical intervals. It also defines a type `boolset`, a specialization of `std::bitset`, used by interval comparison functions.

2 An implementation shall instantiate the template `interval` for the basic floating point types `float`, `double`, and `long double`. It may provide instantiations for other floating point types, such as arbitrary-precision arithmetic.

3 **Interval arithmetic** (IA) is a basic tool for certified mathematical computations. A mathematical interval  $[a, b]$  is the set of numbers lying between the two bounds  $a$  and  $b$ . An `interval` object represents a mathematical interval. The most important use is in computing proven inclusions of the range of a function  $f$ . Such an inclusion is a set containing all the values  $f$  takes as its arguments vary over given input intervals. R. Moore’s **Fundamental Theorem** (1965), the basis of the theory, asserts that if the elementary functions (including arithmetic operations) are given interval versions that possess this inclusion property, and if a function  $f$  is built from elementary functions, then the corresponding interval version of  $f$  possesses the inclusion property, for any interval inputs that do not cause an exception to occur.

A chief reason for choosing the containment set (cset) abstract model, below, is that cset-based interval systems provide exception-free evaluation of numeric functions, while giving the same result as traditional interval systems on inputs that the latter process successfully. Put another way, when “interval version” is replaced by “interval cset version” in the statement of Moore’s theorem, the phrase “for any interval inputs that do not cause an exception to occur” can be deleted.

4 Any implementation of interval arithmetic must be based on an abstract interval model (abstract model for short), which defines the following on an abstract “exact arithmetic” machine. (i) the underlying number system (*whole line*); (ii) the set  $I$  of intervals supported by the model (e.g., are infinite intervals in  $I$ ? is the empty set in  $I$ ?); (iii) the definition (*the mathematical result*) of the arithmetic operations and other supported elementary functions acting on members of  $I$ , including behaviour at special points such as singularities.

5 **Mission.** This standard supports just one abstract model:

- The whole line is the extended reals  $\mathbb{R}^* = \mathbb{R} \cup \{-\infty, +\infty\}$ .
- The set  $I$  comprises all nonempty closed intervals in  $\mathbb{R}^*$ , that is  $[a, b]$  for arbitrary  $a, b \in \mathbb{R}^*$  with  $a \leq b$ , together with the empty set. This includes the whole line, since it equals  $[-\infty, +\infty]$ .

- Arithmetic operations and elementary functions are defined in the *containment set* (cset) sense [18, 5] as follows. For points  $x$  and  $y \in \mathbb{R}^*$  and an arithmetic operation  $\bullet$ , the cset value of  $x \bullet y$  is the set of all possible limits of  $z_r = x_r \bullet y_r$ , where  $x_r \rightarrow x$  and  $y_r \rightarrow y$  as  $r \rightarrow \infty$ , and  $x_r \bullet y_r$  is defined in the ordinary real sense for all  $r$ . For subsets  $X$  and  $Y$  of  $\mathbb{R}^*$ , the cset value of  $X \bullet Y$  is the union of the cset values of  $x \bullet y$  over all  $x \in X$  and  $y \in Y$ .

When  $X$  and  $Y$  are intervals in  $\mathbb{R}^*$ , the mathematical result  $X \bullet Y$  in the sense of clause 4 above (synonym: the *interval cset value*) is the smallest member of  $I$  containing the cset value of  $X \bullet Y$ .

The same notions for elementary functions  $e(x, \dots)$  are defined analogously.

6 For each supported type  $T$ , the implementation-dependent set  $I_T$  of intervals supported by the `interval<T>` instantiation must be a subset of  $I$  that includes the empty set and the whole line.  $I_T$  must be defined in the implementation’s user documentation.

7 **Requirement:** It must be impossible for code (assuming it does not violate the encapsulation provided by `interval` class methods) to create an `interval<T>` object that does not represent a member of  $I_T$ . An implementation that permits such creation is incorrect.

8 **Notation.** `empty` and `whole` denote the `interval` objects (of type  $T$  given by the context) representing the empty set and whole line. `infinity` denotes the representation of  $\infty$ , that is the value `std::numeric_limits<T>::infinity()` if type  $T$  supports it.

Unless stated otherwise,  $X, Y, \dots$  denote the mathematical intervals represented by the `interval` objects `X, Y, \dots`, etc. Similarly,  $e$  is the mathematical function represented by an implemented function `e`. If it is stated that  $X$  encloses  $Y$ , this should be taken to mean that  $X$  encloses  $Y$ , and so on.

9 **Inclusion property.** In the `interval<T>` instantiation, the result of  $X \bullet Y$  or  $e(X, \dots)$ , where  $X, Y, \dots$  are intervals, must be a member of  $I_T$  that contains the mathematical result of  $X \bullet Y$  or  $e(X, \dots)$ .

10 **Notes:**

1. The standard does not specify that any computed inclusion be the smallest (tightest) possible, regarding this as a quality-of-implementation (QOI) issue. However, when a certain tightness is normally achievable, especially when users might find it counter-intuitive not to do so, this is mentioned. For example, with the basic arithmetic operations,  $X \bullet Y$  is expected to be the tightest enclosure of the mathematical result  $X \bullet Y$ . However, this is a QOI issue, and an implementation may trade tightness for speed.
2. The standard does not specify the internal data representation, e.g. “lower/upper bound” or “midpoint/radius”, see 26.6.A.
3. This standard explicitly does not support various alternative models such as: W. Kahan’s interval arithmetic with a single unsigned infinity; or models that support wrap-around intervals such as  $[2, -1]$ , which denotes the unbounded set  $\{x \mid x \leq -1 \text{ or } x \geq 2\}$ ; or Kaucher’s oriented intervals.

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

```
namespace std {
// forward declarations:
template <class T> class interval ;
// 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>&);
// extended division function
template <class T> std::pair<interval<T>, interval<T>>
    xdiv (const interval<T>& lhs, const interval<T>& rhs);
// boolset type
typedef const std::bitset<2> boolset;
// boolset nonmember functions
bool possibly (boolset)
bool certainly (boolset)
bool definitely (boolset)
// comparison operators
template <class T> boolset operator == (const interval<T>&, const interval<T>&);
template <class T> boolset operator != (const interval<T>&, const interval<T>&);
template <class T> boolset operator < (const interval<T>&, const interval<T>&);
template <class T> boolset operator <= (const interval<T>&, const interval<T>&);
template <class T> boolset operator > (const interval<T>&, const interval<T>&);
template <class T> boolset operator >= (const interval<T>&, const interval<T>&);
// values:
template <class T> T inf (const interval<T>&);
template <class T> T sup (const interval<T>&);
template <class T> interval<T> mag (const interval<T>&);
template <class T> interval<T> mig (const interval<T>&);
template <class T> T midpoint (const interval<T>&);
template <class T> T width (const interval<T>&);
// algebraic and transcendental functions from <cmath> library:
template <class T> interval<T> abs (const interval<T>&); //interval form of fabs!
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> atan2 (const interval<T>& , const interval<T>&);
template <class T> interval<T> ceil (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> floor (const interval<T>&);
template <class T> interval<T> fmod (const interval<T>& , 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>& , const interval<T>&);
template <class T> interval<T> pow (const interval<T>& , int);
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>&);
template <class T> interval<T> square (const interval<T>&); //not in <cmath>!
template <class T> interval<T> tan (const interval<T>&);
template <class T> interval<T> tanh (const interval<T>&);
// set operations:
template <class T> bool equal(const interval<T>&, const interval<T>&);
template <class T> bool subseteq (const interval<T>&, const interval<T>&);
template <class T> bool subseteq (const T&, const interval<T>&);
template <class T> bool interior (const interval<T>&, const interval<T>&);
template <class T> bool interior (const T&, const interval<T>&);
template <class T> bool disjoint (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>&);
// I/O 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>&);

} // 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 char *);
interval (const T& t);
interval (const T& lo, const T& hi);
template <class U> explicit interval (const U&);
template <class U> interval (const U&, const U&);
template <class U> interval (const interval <U>&);
bool is_empty_interval() const ;
T lower() const ;
T upper() const ;
interval & operator =(const T&);
interval & operator +=(const T&);
interval & operator -=(const T&);
interval & operator *=(const T&);
interval & operator /=(const T&);
template <class U> interval & operator =(const interval <U>&);
template <class U> interval & operator +=(const interval <U>&);
template <class U> interval & operator -=(const interval <U>&);
template <class U> interval & operator *=(const interval <U>&);

```

```

template <class U> interval & operator /=(const interval <U>&);
static interval whole();
static interval empty();
static interval pi();
};
}
// of namespace std

```

**1 Requirement:** This template does not specify how the data defining an `interval` object is stored. However, to allow the `valarray` class to be used with intervals, it is required that this data occupy a contiguous block in memory. If the base type `T` is of fixed size (such as the built-in types `float`, `double`, and `long double`), the size of an `interval<T>` object shall also be of fixed size. See 26.6.3 clause 5, which creates a connection between the precision of an interval and that of its base type.

### 26.6.3 Interval member functions [lib.interval.members]

```
interval();
```

**1 Effects:** Constructs `whole`.

**2 Notes:** `whole` is considered a safer default `interval` value than `empty`, since the latter is more likely to cause faulty code to “lie”.

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

**3 Effects:** Constructs an enclosure of the external interval represented by the null-terminated byte string `s` pointed to by `s`. The whole of `s` must represent an external interval according to 26.6.14. Raises an exception if this is not the case. [*Begin example: s = “[1.2;3.4]” (with any amount of white space around the tokens) is OK, while “[1.2; 3.4] abc” and “[1.2; 3.4] [5.6; 7.8]” cause an exception. —end example.*]

```
interval (const T& x);
```

**4 Effects:** Identical to the effects of `interval(x,x)`.

**5 Requirement:** If `x` is a finite value of type `T` then, after executing `xi=interval(x)`, calls to `xi.lower()` and `xi.upper()` shall each return `x` exactly. That is, singleton intervals are stored to at least the precision of their base type. See also §26.6.2 clause 1.

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

**6 Effects:** If `lo` and `hi` hold numbers  $a$  and  $b$ , this constructs an  $I_T$ -enclosure `X` of the mathematical set  $X = \{x \in \mathbb{R}^* \mid a \leq x \text{ and } x \leq b\}$ , otherwise raises an exception.

**7 Notes:** This description has the following cases and consequences.

1. An exception is raised if, and only if, `lo` or `hi` is not a number (NaN, on IEEE 754 systems).
2. It is not mandatory for an implementation to support infinite intervals except for `whole`.
3. `X` is empty if  $a > b$ . In this case, `empty` must be returned.
4. If the implementation uses the lower/upper bound representation, then `X` is expected to represent  $X$  exactly whenever  $X$  is defined. This is a QOI issue. With the midpoint/radius representation, `X` cannot necessarily represent  $X$  exactly, for instance if `lo` and `hi` hold two consecutive `T`-numbers.

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

**8 Effects:** Identical to those of `interval(x,x)`.

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

9 **Effects:** Identical to the effects of constructing the corresponding `interval<U>` object and converting it to `interval<T>`, that is to doing `interval<T>(interval<U> (lo,hi))`. In particular, an exception is raised if, and only if, `lo` or `hi` is not a number.

```
template <class U> interval (const interval <U>& XU);
```

10 **Effects:** Constructs an interval `X` enclosing the interval `XU`.

11 **Notes:** `T` and `U` are two floating point types, possibly the same. This has the following cases and consequences.

1. If `T` and `U` are the same, `X` must equal `XU`.
2. If `U` is of a lower precision than `T`, meaning that every `U`-number is also a `T`-number, then `X` should represent the same mathematical interval as `XU`. This is a QOI issue.
3. Otherwise, it is possible that the bounds of `XU` are beyond the `T`-representable range, so that a finite interval may be converted to an infinite one.
4. This function never throws an exception.

```
bool is_empty_interval() const;
```

12 **Returns:** true if `*this` equals `empty`, else false.

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

13 **Returns:** a lower bound of the interval represented by `*this`.

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

14 **Returns:** an upper bound of the interval represented by `*this`.

15 **Notes:** These apply to `lower()` and `upper()`.

1. The exact lower or upper bound should be returned if it is exactly `T`-representable. This is a QOI issue. In general, a bound is not exactly `T`-representable if the midpoint/radius data representation is used.
2. The result is implementation-defined if the bound is infinite and the number system does not support  $\infty$ .
3. The result is implementation-defined if `*this` is empty. Compare `inf()` and `sup()`, which return  $+\infty$  and  $-\infty$  on the empty set.

## 26.6.4 Interval member operators [lib.interval.members.ops]

```
template <class U> interval<T>& operator =(const interval <U>& XU);
```

1 **Effects:** Identical to those of converting `XU` to `interval<T>` and then assigning. That is, `X = XU`; has the same effect as `X = interval<T>(XU)`;

2 **Returns:** `*this`.

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

3 **Effects:** Let `*this` and `rhs` represent the intervals `X` and `Y` on entry. Stores in `*this`, and returns an enclosure of the interval cset value (mathematical result) `X + Y`.

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

4 **Effects:** Let `*this` and `rhs` represent the intervals `X` and `Y` on entry. Stores in `*this`, and returns an enclosure of the interval cset value (mathematical result) `X - Y`.

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

5 **Effects:** Let `*this` and `rhs` represent the intervals  $X$  and  $Y$  on entry. Stores in `*this`, and returns an enclosure of the interval cset value (mathematical result)  $X \times Y$ .

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

6 **Effects:** Let `*this` and `rhs` represent the intervals  $X$  and  $Y$  on entry. Stores in `*this`, and returns an enclosure of the interval cset value (mathematical result)  $X/Y$ .

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

7 **Effects:** Identical to those of `*this += interval<T>(rhs)`.

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

8 **Effects:** Identical to those of `*this -= interval<T>(rhs)`.

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

9 **Effects:** Identical to those of `*this *= interval<T>(rhs)`.

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

10 **Effects:** Identical to those of `*this /= interval<T>(rhs)`.

## 26.6.5 Interval non-member operations [lib.interval.ops]

### Unary operators.

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

1 **Returns:**  $X$ .

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

2 **Returns:** An enclosure of the mathematical result  $-X$ . If  $T$  is symmetric, that is if  $-x$  is a  $T$ -number whenever  $x$  is so, then this should return  $-X$  exactly.

### Binary operators.

```
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` in the above three cases.

```
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);
```

4 **Returns:** `interval<T>(lhs) -= rhs` in the above three cases.

```
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);
```

5 **Returns:** `interval<T>(lhs) *= rhs` in the above three cases.

```
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 in the above three cases.
```

### 26.6.6 Extended division function [lib.interval.xdiv]

```
template <class T> std::pair<interval<T>, interval<T>> xdiv (const interval<T>& lhs,
const interval<T>& rhs);
```

1 **Returns:** Let `lhs` and `rhs` represent intervals  $X$  and  $Y$ . If  $Y$  contains zero, the (non-interval) cset value  $X/Y$  can be the union of two disjoint intervals  $V = [-\infty, v]$  and  $W = [w, +\infty]$ . In this case, `xdiv` returns a pair containing  $V$  and  $W$  in this order. Otherwise, returns a pair whose first member is the single-interval cset result and whose second member is the empty interval.

2 **Notes:** When the non-interval cset value is the union of  $V$  and  $W$  as above, the interval cset result of division  $X/Y$  is `whole`, resulting in a loss of information. `xdiv` avoids this information loss.

### 26.6.7 boolset type and functions [lib.interval.boolset]

1 **Notes:** Defines a type representing subsets  $B$  of  $\{\text{false}, \text{true}\}$  and three nonmember functions. A `boolset` value `bs` representing  $B$  has two boolean fields accessible as `bs[0]` and `bs[1]`. By definition, `bs[0]` is true if (and only if)  $(\text{false} \in B)$ , while `bs[1]` is true if  $(\text{true} \in B)$ .

```
typedef const std::bitset<2> boolset;
```

```
bool possibly (boolset bs)
```

2 **Returns:** True if  $\text{true} \in B$ , that is if `bs[1]` is true.

```
bool certainly (boolset bs)
```

3 **Returns:** True if  $\text{false} \notin B$ , that is if `bs[0]` is false.

```
bool definitely (boolset bs)
```

4 **Returns:** True if  $B = \{\text{true}\}$ , that is if `bs[0]` is false and `bs[1]` is true.

5 **Notes:** `certainly` and `definitely` behave differently when an argument to a comparison is empty. Let  $X$  and  $Y$  be intervals and  $\bullet$  be one of the comparison operators in 26.6.8.

Then `possibly(X  $\bullet$  Y)` is true if  $x \bullet y$  is true for some  $x \in X$  and  $y \in Y$  — which implies  $X$  and  $Y$  are nonempty.

`certainly(X  $\bullet$  Y)` is true if  $x \bullet y$  is true for all  $x \in X$  and  $y \in Y$  — which by set theory convention is true if one or both of  $X$  or  $Y$  is empty.

`definitely(X  $\bullet$  Y)` is true if  $X$  and  $Y$  are nonempty and  $x \bullet y$  is true for all  $x \in X$  and  $y \in Y$ , equivalent to `possibly(X  $\bullet$  Y) && certainly(X  $\bullet$  Y)`.

*[Begin example:*

`possibly(X < Y)` means  $X$  and  $Y$  are nonempty and some  $x \in X$  is  $<$  some  $y \in Y$ .

`certainly(X < Y)` means there is no element of  $X$  that is  $\geq$  an element of  $Y$ , but either of  $X$  or  $Y$  might be empty.

`definitely(X < Y)` means  $X$  and  $Y$  are nonempty and  $X$  is entirely to the left of  $Y$ .

`definitely(X != Y)` means  $X$  and  $Y$  are “comparable” in the sense that they are nonempty and  $X$  is either entirely  $<$   $Y$  or entirely  $>$   $Y$ .

`certainly(X != Y)` returns true if  $X$  and  $Y$  are comparable in the above sense, but also if either is empty.

`definitely(X == X)` is equivalent to  $X$  being a singleton. —end example.]

## 26.6.8 Interval comparison operators [lib.interval.compare]

```
template <class T> boolset operator ==(const interval<T>& X, const interval<T>& Y);
template <class T> boolset operator !=(const interval<T>& X, const interval<T>& Y);
template <class T> boolset operator < (const interval<T>& X, const interval<T>& Y);
template <class T> boolset operator <=(const interval<T>& X, const interval<T>& Y);
template <class T> boolset operator > (const interval<T>& X, const interval<T>& Y);
template <class T> boolset operator >=(const interval<T>& X, const interval<T>& Y);
```

1 **Returns:** Let  $\bullet$  stand for one of the above operators. Returns a `boolset` value representing the set  $X \bullet Y = \{x \bullet y \mid x \in X \text{ and } y \in Y\} \subseteq \{\text{false}, \text{true}\}$ .

2 **Notes:** This definition gives the correct cset value for infinite intervals automatically.

If  $X$  or  $Y$  is empty, the result is the `boolset` value representing empty.

This definition overrides the standard meaning of the operators `==` and `!=`.

## 26.6.9 Interval value operations [lib.interval.value.ops]

1 **Notes:** With the bounds data representation (but not with the midpoint/radius representation), `inf`, `sup`, `mig`, and `mag` are expected to return the exact value for any input. With the midpoint/radius representation (but not with the bounds representation), `midpoint` and `width` are expected to return the exact value for any input. These are QOI issues.

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

2 **Returns:** `X.lower()` when  $X$  is not empty. Otherwise `infinity` or an implementation-defined value if  $T$  does not support  $\infty$ .

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

3 **Returns:** `X.upper()` when  $X$  is not empty. Otherwise `-infinity` or an implementation-defined value if  $T$  does not support  $\infty$ .

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

4 **Returns:** An upper bound of the magnitude of interval  $X$ . Equivalent to `sup(abs(X))`. In particular if  $X$  is empty, returns `-infinity` or an implementation-defined value if  $T$  does not support  $\infty$ .

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

5 **Returns:** A lower bound of the magnitude of interval  $X$ . Equivalent to `inf(abs(X))`. In particular if  $X$  is empty, returns `infinity` or an implementation-defined value if  $T$  does not support  $\infty$ .

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

6 **Returns:** An approximation to the midpoint  $(a + b)/2$ , if  $X$  represents a nonempty finite interval  $[a, b]$ . Otherwise `NaN` or an implementation-defined value if  $T$  does not support `NaN`.

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

7 **Returns:** An upper bound to  $b - a$ , if  $X$  represents a nonempty interval  $[a, b]$ . If  $X$  is empty then `-infinity` or an implementation-defined negative value if  $T$  does not support  $\infty$ .

## 26.6.10 Interval mathematical functions [lib.interval.math]

1 Interval cset versions of the functions in the following list will be provided for each supported type  $T$ . They comprise all functions in the `<cmath>` library *except* `frexp`, `ldexp`, `modf`, and `fabs`. Floating

point `fabs` is the absolute value function: the interval version is named `abs` in this standard. That is, `abs(X)` is an enclosure of  $\{|x| \mid x \in X\}$ .

Also included is the `square` function:

$$\text{square}(x) = x^2 \quad (x \in \mathbb{R}).$$

Reference implementations following the cset model of this standard are provided e.g. by `flib++` and by the C++ compiler of Sun Microsystems.

[JDP Note: I felt the excluded functions, above, have no need for interval versions, but the rest do. ISL team, please check through the list and include/exclude items, with reasons.

GFC: OK by me.]

```
namespace std {

    // algebraic and transcendental functions:
    template <class T> interval<T> abs (const interval<T>&); //interval form of fabs!
    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> atan2 (const interval<T>& , const interval<T>&);
    template <class T> interval<T> ceil (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> floor (const interval<T>&);
    template <class T> interval<T> fmod (const interval<T>& , 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>& , const interval<T>&);
    template <class T> interval<T> pow (const interval<T>& , int);
    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>&);
    template <class T> interval<T> square (const interval<T>&); //not in <cmath>!
    template <class T> interval<T> tan (const interval<T>&);
    template <class T> interval<T> tanh (const interval<T>&);

} // of namespace std
```

2 The interval cset value (the mathematical result, 26.6 clause 4) of each function  $e$  on given interval argument(s) is uniquely defined by the definition of  $e$  as a real-valued function of real argument(s). For the above functions, this is defined in the specification of `math.h` in 7.12 of the C standard, but see exceptions and clarifications below.

3 The standard only requires that the computed result enclose the mathematical result for all arguments. Tightness of enclosure is a QOI issue.

4 The definition of cset implies that *loose evaluation* is to be used: out-of-domain points within an interval argument do not cause an exception. Namely, let function  $e$ , with domain  $\mathcal{D}_e \subseteq \mathbb{R}$ , be cset-evaluated at interval argument  $X \subseteq \mathbb{R}^*$ . Points  $x \in X$  that are outside the closure of  $\mathcal{D}_e$  have an empty cset, so are in effect ignored. Points outside  $\mathcal{D}_e$ , but on its boundary, always contribute at least one value defined as a limit.

For example, the domain of  $\log x$  is the open interval  $(0, \infty)$ . If  $X = [-4, 2]$ , the interval cset value of  $\log(X)$  is  $[-\infty, \log 2]$ , where those  $x \in X$  such that  $x < 0$  are outside the closure of  $\mathcal{D}_e$  and

contribute nothing to the result; the point  $x = 0$  is on the boundary of  $\mathcal{D}_e$  and contributes  $-\infty$ ; and those  $x$  with  $x > 0$  contribute  $(-\infty, \log 2]$ .

5 Implementers should consider providing a global DISCONTINUOUS flag, which takes the place of an “out-of-domain exception”. For its rationale and definition, see [18].

6 **Clarifications.** The C standard, Appendix F.9, specifies the values of various standard functions at singularities and other special points. In many cases, these coincide with the cset value, but in cases of conflict, the cset definition is to be followed. In particular:

- **Signed zeros.** There are no separate  $-0, +0$  in the number system  $\mathbb{R}^*$  used by this standard, hence the definitions in C standard Appendix F.9 based on this distinction do not apply.
- **Power functions.** The function `interval pow(interval, interval)` is the interval version of the real two-variable function  $f(x, y) = x^y = \exp(y \log x)$ , defined on the domain  $x, y \in \mathbb{R}, y > 0$ . The function `interval pow(interval, int)` is the interval version of the *family* of real one-variable functions  $f_n(x) = x^n$  for any integer  $n$ , each defined on the whole of  $\mathbb{R}$ .
- **atan2.** The straightforward definition of `atan2(y, x)` is as the Principal Value: a one-valued real function defined everywhere except at  $(0, 0)$ , discontinuous at the branch cut along the negative real axis where the value is either  $-\pi$  or  $\pi$  — the C standard (7.12.4.4) does not say which. The cset value on the branch cut is the two points  $\{-\pi, \pi\}$ , hence the interval cset value, for any arguments that define a rectangle intersecting the branch cut, is the interval  $[-\pi, \pi]$ .

Consideration should be given to a more “intelligent” interval version that takes account of the multi-valued, spiral nature of the graph of `atan2`, when given its general definition: `atan2(y, x)` = the set of all  $\theta$  such that  $r \cos \theta = x$  and  $r \sin \theta = y$  for some  $r > 0$ . This would be useful, e.g., for reliable computations of winding numbers. For some work on this, see Walster [22].

### 26.6.11 Interval set operations [lib.interval.set.ops]

```
template <class T> bool equal(interval<T> const & X, interval<T> const & Y);
```

1 **Returns:** true if  $X$  and  $Y$  are equal as sets, else false .

2 **Notes:** Differs from the operator `==` in the semantics (C++ standard, 5.10).

```
template <class T> bool subseteq (const interval<T>& X, const interval <T>& Y);
```

3 **Returns:** true if  $X \subseteq Y$  in the set sense, else false .

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

4 **Returns:** Equivalent to `subseteq(interval<T>(lhs), rhs)`.

```
template <class T> bool interior (const interval<T>& X, const interval <T>& Y);
```

5 **Returns:** true if  $X$  is contained in the interior of  $Y$  in the topological sense, else false .

6 **Notes:** That is (whether the intervals are finite or infinite), true if either  $X$  is empty, or  $\underline{y} < \underline{x}$  and  $\bar{x} < \bar{y}$ , where  $X = [\underline{x}, \bar{x}]$  and  $Y = [\underline{y}, \bar{y}]$ .

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

7 **Returns:** Equivalent to `interior(interval<T>(lhs), rhs)`.

```
template <class T> bool disjoint (const interval<T>& X, const interval<T>& y);
```

8 **Returns:** true if  $X \cap Y$  is empty, else false .

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

9 **Returns:** An enclosure of  $X \cap Y$ .

10 **Notes:** With the lower/upper bound representation, it should be possible to return the exact intersection. This is not always possible with the midpoint/radius representation.

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

11 **Returns:** An enclosure of  $X \cup Y$ .

12 **Notes:** With the lower/upper bound representation, it should be possible to return the exact interval hull of  $X \cup Y$ . This is not always possible with the midpoint/radius representation.

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

13 **Returns:** a pair of interval objects enclosing (in that order) the intervals  $Y = \{x \in X \mid x \leq t\}$  and  $Z = \{x \in X \mid x \geq t\}$ .

14 **Notes:** With the lower/upper bound representation, it should be possible to return  $Y$  and  $Z$  exactly. This is not always possible with the midpoint/radius representation.

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

15 **Returns:** `split(X, midpoint(X))` if  $X$  is not empty, a pair of empty intervals otherwise.

### 26.6.12 Interval static value operations [lib.interval.static.value.ops]

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

1 **Returns:** whole.

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

2 **Returns:** empty.

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

3 **Returns:** An enclosure of the exact value  $\pi$ .

### 26.6.13 Interval I/O operations [lib.interval.io]

[JDP Note: I would like to insert two low-level functions. One to convert a  $T$  number  $x$  to external form as a (decimal) string  $s$ , with arguments to specify number of significant digits, and whether rounding up or down. The other to do essentially the reverse. This will make it significantly easier, e.g., for users to implement more sophisticated I/O functions for intervals.

GFC: Agree]

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

1 **Effects:** An external interval  $X$ , see 26.6.14, is read off the stream `is`. An enclosure of the mathematical interval represented by  $X$  is stored in  $X$ .

If bad input is encountered, the function calls `is.setstate(ios::failbit)` (which may throw `ios::failure` 27.4.4.3).

2 **Notes:** Ideally  $X$  should be the smallest  $I_T$ -enclosure of  $X$ . This is a QOI issue.

3 **Returns:** `is`.

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

4 **Effects:** Constructs an external interval  $X$ , see 26.6.14, enclosing  $X$ , and writes it to the stream `os`.

5 **Notes:** [Needs a description of the features offered by `traits`. This lets one specify the number of significant figures? locale info? ...]

6 **Returns:** `os`.

## 26.6.14 Interval I/O conversion [lib.interval.ioconversion]

1 An *external interval* is a string  $X$  that represents a mathematical interval, according to the following rules. First  $X$  is preprocessed into tokens, and white-space discarded, following standard C++ rules. Valid tokens are integral or floating literals, the characters `[`, `]`, `;`, `+`, `-`, and the (case insensitive) identifiers `inf`, `empty`, and `whole`. The following meanings are assigned:

Token combination	Meaning
<code>- inf</code>	$-\infty$
<code>inf</code> or <code>+ inf</code>	$+\infty$
<code>[ empty ]</code> or <code>[ ]</code>	The empty set
<code>[ whole ]</code>	The whole line
<code>[ a ; b ]</code> where $a$ and $b$ are integral or floating literals or one of the combinations meaning $\pm\infty$ .	The mathematical interval $X = [a, b]$ where $a$ and $b$ are the exact values denoted by $a$ and $b$ . In particular $X$ is empty if $a > b$ .
<code>[ a ]</code>	The same as <code>[ a ; a ]</code>

The spaces in the left hand column merely serve to delimit tokens and are not necessary in the input string.

2 **Inclusion property.** On input, an external interval  $X$  must be converted to an `interval` object  $X$  that encloses it. (More precisely, the interval represented by  $X$  must enclose the interval represented by  $X$ .) On output, an interval object must be converted to an external interval that encloses it. Tightness of the enclosures is a QOI issue.

3 A string that cannot be parsed as an external interval shall not be given any default `interval` value (e.g. `whole`). Hence an input routine on encountering such a string must throw an exception.

4 **Notes:** An implementation may provide features beyond the above, e.g.:

1. Simple expressions in the input stream and symbolic representations of constants such as  $\pi$ , so that for instance “[2\*pi]” would be converted to an interval guaranteed to enclose  $2\pi$ .
2. Floating literals in an octal or hexadecimal base, allowing exact conversion between internal and external intervals, on both input and output.

## 26.6.A Design notes (informative)

To be written over a period, with sections aiming to cover the following. Links to other sources can be given.

### 26.6.A.1 Tables of interval cset values

For basic arithmetic operations (BAOs) ( $+\infty/+\infty = [0, +\infty]$  etc.) and standard functions ( $\sin(+\infty) = [-1, 1]$  etc.).

### 26.6.A.2 Pseudo-code for the BAOs

Code that is considered efficient on typical current machines, with alternatives for different data representations.

### 26.6.A.3 The DISCONTINUOUS flag

Definition, suggestions for implementation, possible pitfalls in use and ways round them.

**26.6.A.4 Guidelines for unit testing**

(a) “logic” tests; (b) “inclusion” tests.

**26.6.A.5 Test cases for unit tests of each function**

**26.6.A.6 etc...**

## 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)
{
    typedef interval<T> I;
    T feps = std::numeric_limits<T>::min();
    I eps(-feps, feps);

    I y = f(current);

    // Evaluate f() over current interval.
    // Short circuit if current does not contain a zero of f
    if (! subseteq(T(0), y)) return oit;

    // Stop the dichotomy if res is small enough - this probably prevents useless work.
    // Also stop if we have reached the maximal precision.
    if (subsetq(y, eps) || 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(width(ip.first)==0 || width(ip.second)==0) {
        *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;9.3132257461547851562 e -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:

*The system of polynomials is missing*

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` over components) and that `assign_box(r,epsilon)` assigns `epsilon` to every component of the vector `r`.

```

// Returns a set of boxes (sub-boxes 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;
  T feps = std::numeric_limits<T>::min();
  I eps(-feps, feps);
  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 probably prevents 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, eps);
  if (subsepeq(res, r) || 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 (width(begin[s])==0 || width(end[s])==0) {
        *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]
[ -0.822875672187304497; -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 many people for their comments on earlier versions of this proposal: Lawrence Crowl and his colleagues at Sun; John Pryce's colleagues George Corliss, Baker Kearfott, Ned Nedialkov and Spencer Smith from the Interval Subroutine Library project; and many others including Stefan Schirra, Joris Van Der Hoeven, Bernard Mourrain.

Finally, thanks to the Library Working Group for the comments and positive feedback on the initial version presented in Mont-Tremblant.

## References

- [1] Herve Brönnimann, Guillaume Melquiond, and Sylvain Pion, *The design of the Boost interval arithmetic library*, Theoretical Computer Science, Special Issue on Real Numbers and Computers (RNC5) (accepted), Preprint available at <http://photon.poly.edu/~hbr/publi/boost-interval-rnc5/tcs.pdf>.
- [2] CGAL, *CGAL. Computational Geometry Algorithms Library*, <http://www.cgal.org/>.
- [3] Samuel P. Ferguson and Thomas F. Hales, *A formulation of the Kepler conjecture and other papers*, 1998, <http://www.math.pitt.edu/~thales/kepler98/>.
- [4] Frederic Goualard, *Gaol, not just another interval library*, 2006, <http://www.sourceforge.net/projects/gaol/>.
- [5] Eldon Hansen and G. William Walster, *Global optimization using interval analysis, 2nd edition*, Marcel Dekker, New York, 2005.
- [6] T. Hickey, Q. Ju, and M. H. Van Emden, *Interval arithmetic: From principles to implementations*, J. ACM **48** (2001), no. 4, 1038–1068.
- [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, Heidelberg, 2001.
- [8] W. M. Kahan, *A more complete interval arithmetic*, Lecture notes for a summer course at the University of Michigan, University of Michigan, Ann Arbor, 1968.
- [9] R. Baker Kearfott, *Interval Computations: Introduction, uses and resources*, 1996, [ftp://interval.louisiana.edu/pub/interval\\_math/papers/papers-of-Kearfot%t/Euromath\\_bulletin\\_survey\\_article/survey.ps](ftp://interval.louisiana.edu/pub/interval_math/papers/papers-of-Kearfot%t/Euromath_bulletin_survey_article/survey.ps).
- [10] R. Baker Kearfott, Keith Bierman, George F. Corliss, David Hough, Andrew Pitonyak, Michael Schulte, G. William Walster, and Wolfgang Walter, *A specific proposal for interval arithmetic in Fortran*, Tech. report, University of Louisiana, Lafayette, 1996, [interval.louisiana.edu/F90/f96-pro.asc](http://interval.louisiana.edu/F90/f96-pro.asc).
- [11] R. Baker Kearfott and Vladik Kreinovich (eds.), *Applications of interval computations*, Kluwer, 1996.
- [12] O. Knueppel, *PROFIL/BIAS – A fast interval library*, Computing **53**, no. 3–4, 277–287, <http://www.ti3.tu-harburg.de/knueppel/profil/>.

- [13] Vladik Kreinovich, *Interval Computations*, <http://www.cs.utep.edu/interval-comp/main.html>.
- [14] Branimir Lambov, *Interval arithmetic using SSE2 (draft)*, Tech. report, BRICS, University of Aarhus, Denmark, January 2006.
- [15] M. Lerch, G. Tischler, J. Wolff von Gudenberg, W. Hofschuster, and W. Krämer, *The interval library filib++ 2.0 - design, features and sample programs.*, Preprint 2001/4, Universität Wuppertal, Wuppertal, Germany, 2001.
- [16] Sun Microsystems, *C++ interval arithmetic programming reference*, <http://docs.sun.com/db/doc/806-7998>.
- [17] Ramon E. Moore, *Interval analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1966.
- [18] John D. Pryce and George F. Corliss, *Interval arithmetic with containment sets*, Computing, Technical Report DoIS/TR01/06, Department of Information Systems, Shrivenham Campus, Cranfield University, Swindon SN6 8LR, UK, 2005. Available at <http://homepage.ntlworld.com/j.d.pryce/isloct05/prycecorliss06.pdf>.
- [19] Nathalie Revol and F. Rouillier, *MPFI 1.0, Multiple precision floating-point interval library*, [http://www.ens-lyon.fr/~nrevol/mpfi\\_toc.html](http://www.ens-lyon.fr/~nrevol/mpfi_toc.html).
- [20] Siegfried M. Rump, *INTLAB interval toolbox, version 5.2*, 1999–2006, <http://www.ti3.tu-harburg.de/intlab.ps.gz>.
- [21] G. William Walster, *The extended real interval system*, 1998.
- [22] ———, *Interval angles and the Fortran ATAN2 intrinsic function*, 2002, [http://developers.sun.com/prodtech/cc/articles/int\\_angles/interval-angl%es.pdf](http://developers.sun.com/prodtech/cc/articles/int_angles/interval-angl%es.pdf).