Vienna proposal for interval standardization

Arnold Neumaier

Fakultät für Mathematik, Universität Wien
Nordbergstr. 15, A-1090 Wien, Austria
email: Arnold.Neumaier@univie.ac.at
WWW: `http://www.mat.univie.ac.at/~neum`

with improvements suggested by remarks of

Bo Einarsson, Maarten van Emden,
Michel Hack, Nate Hayes, Ulrich Kulisch,
Jens Maurer, Ian McIntosh, Sylvain Pion,
John Pryce, Hermann Schichl,
Wolff von Gudenberg,
Paul Zimmermann, and Dan Zuras

Final version, December 19, 2008

Abstract. This document contains a detailed proposal for a future IEEE 1788 standard on interval arithmetic. It is written in a form that should be not too difficult to transform into a formal, complete and fully precise document specifying the standard to be.

Part 1 contains a concise summary of the basic assumptions (some of which may be controversial) upon which the remaining document is based. The items in Part 1 are grouped such that separate voting on each issue is meaningful.

Parts 2-5 specify the internal representation of intervals and the operations defined on them. Part 6 specifies the external representation of intervals and the conversion between internal and external representations.

Part 7 is optional and discusses useful modifications of the directed rounding behavior specified in the IEEE 754-2008 standard that would simplify an implementation of the proposed standard.

This final version incorporates many suggestions and corrections by the people mentioned above. In particular, comprehensive discussions with Michel Hack, who read and commented in detail many intermediate versions, had a strong influence on this proposal.

Contents

Part 1. Basic assumptions

1.0. Theme of Part 1

This part summarizes basic assumptions upon which the remaining document is based, grouped such that separate voting on each issue is meaningful.

Interval analysis (see, e.g., the textbooks [2,3]) is the theory and practice of rigorously working on a computer with certain (exactly known) and uncertain (i.e., possibly not exactly known) real numbers, represented as intervals.
It involves the appropriate use of standard floating-point calculations (in round to nearest mode), directed floating-point calculations (in rounding modes up or down), and interval arithmetic, combined in a way that gives reasonable enclosures of the results with an acceptable cost.

Current applications of interval analysis include global optimization (used in lots of different applications), solving equations with multiple solutions (e.g., in chemical engineering, computational geometry, robotics), quantified constraint satisfaction (e.g, finding safe work spaces for robots), design under uncertainty, computer-assisted proofs. Many applications (e.g., all global problems) need good bounds for very wide intervals, some (e.g., rounding error control) need high accuracy for narrow intervals.

Interval arithmetic is the collection of operations with intervals underlying interval analysis, and does not comprise directed floating-point calculations per se; however, its implementation would benefit from the availability of directed floating-point arithmetic with certain properties (spelled out in Part 7 containing optional recommendations).

The proper use of interval arithmetic requires attention to details beyond those of usual numerical computations since otherwise the results may be useless, either being invalid (i.e., not enclosing the true results) or overly pessimistic. Thus, interval arithmetic cannot be regarded in isolation from interval analysis as the body of theory and techniques for making good use of interval arithmetic. Therefore, this proposal is complemented by two other documents describing some

of the interval analysis tools available:
- Computer graphics, linear interpolation, and nonstandard
  intervals [4],
- Improving interval enclosures [5].
See also [1] for standardized notation in interval analysis.

In this proposal, care has been taken to obtain a satisfactory balance
between the need to limit the implementation work and leave implementors
the freedom to add useful features of their own choice, and things
that need to be standardized because of one of
- frequent usage,
- necessity in an important application,
- ambiguities that require standardization,
- options that guarantee compatibility of useful extensions,
and marginally used constructs that can be left to the programmer
since it can be fabricated easily from what is provided if the
construct is needed.

1.1. The number system

This document specifies the interval arithmetic operations deemed
necessary or (for optional features) useful for a satisfactory
computer platform for interval analysis. It restricts attention to
interval arithmetic for certain and uncertain real numbers,
represented as real intervals.

Complex numbers or complex intervals are nowhere discussed in
this document. (Complex intervals are not much used in current
practice. Moreover, it is not difficult to simulate complex interval
arithmetic in software based on available real interval arithmetic.)

All operation symbols in this document denote (unless explicitly said
otherwise) exact operations on real numbers.

The representations of real numbers or pseudo-numbers (Inf, -Inf, NaN)
in the particular language in which the proposed standard
is made available (integers, floating-point numbers, fixed-point
numbers, etc.) are called L-numerals (L for language); there may be
several types of L-numerals, perhaps of different precision or
different radix.

The representations of real numbers denoting bounds of intervals are

called B-numerals (B for bounds, see Section 1.6).
Numerals are either B-numerals or L-numerals; B-numerals may be but need not be among the L-numerals, and L-numerals may be but need not be among the B-numerals.

Finite numerals denote real numbers, which define their value.
Other numerals are classified as +Inf,-Inf, and NaN, depending on whether their value is +Inf, -Inf, or undefined; they do not denote real numbers, no matter how they were generated.
(See Section 2.1 about the values of numerals.)

The terms numeral and B-numeral were chosen to simplify talking about numbers represented in different formats; usually the format hardly matters for the semantics of the specifications. Thus, hardly any demands are made on the format of the numerals. The standard could even be based on exotic formats such as level-index arithmetic that makes overflow and underflow almost impossible.

1.2. The set of intervals

Textbook intervals are closed and connected sets of real numbers, the closed intervals familiar from mathematical textbooks. Nonempty textbook intervals are represented by two bounds, their infimum and their supremum. (See also Section 1.6.)

Standard intervals denote textbook intervals whose infimum and supremum are representable by two B-numerals.
Nonstandard intervals do not denote textbook intervals.

Except in Part 1 and Part 2, where nonstandard intervals are discussed, the unqualified word 'interval' always refers to standard intervals with the above textbook interpretation.

This excludes infinite numbers as element of an interval, which would give unduly pessimistic results in cases such as division by an interval with a zero endpoint; e.g., it gives
[1,2]/[0,1] = Entire in place of [1,Inf] in optimal forward arithmetic.

It also excludes the consideration of arithmetic based on a midpoint-radius representation, which is considered only in conversion between text and interval.

This exclusion is motivated by the fact that in a midpoint-radius representation, it is impossible to represent known uncertainty in the form of inequalities such as x>=1000. Also, optimal rounded interval arithmetic based on the midpoint-radius representation of intervals with real bounds appears to cost twice the amount of work of that needed in the standard representation. The cheaper centered multiplication has the drawback of up to 50% of overestimation of the width.

It also excludes the consideration of arithmetic based on nonstandard intervals (Kahan arithmetic, Kaucher arithmetic, modal arithmetic).

However, certain applications merit providing compatibility of the proposed standard with these not mutually compatible nonstandard variants of interval arithmetic.
The availability of nonstandard intervals allows (but does not force) implementors to extend the functionality of interval arithmetic to handle either Kahan arithmetic, or Kaucher arithmetic, or modal arithmetic consistent with their traditional interpretation.

1.3. Arithmetic operations on intervals

Nullary, unary, and binary interval operations are defined (in Part 3) in a forward mode in such a way that
- they can be used to overload arbitrary arithmetic expressions and yield an enclosure of the range of the function defined by this expression on the set of all arguments inside the corresponding intervals for which the expression makes sense in real arithmetic;
- the result is always a standard interval if the inputs are standard intervals;
- the most basic single operations give the tightest possible enclosure.

In addition, to support correct and optimal constraint propagation (a very important technique in many current applications), there are reverse operations (Sections 3.9 and 3.11) that enclose the set of solutions of equations involving a single operation only, restricted by enclosures of the arguments and results within given intervals.

If these reverse operators were not provided by the standard, they would have to be written separately by anyone using intervals

for constraint propagation, a needless duplication of effort,
with results that are perhaps not best possible.

The above formulation excludes division with a noninterval set result,
which instead is provided as division with gap (Section 5.7).

It also excludes unduly pessimistic arithmetic options for division,
e.g., those giving [0,1]/[0,1] = Entire instead of [0,Inf].

The advantages that csets had in the past for applications to
constraint propagation and in the interval Newton method are
fully conserved by the availability of the forward and the reverse
mode, needed for optimal results in constraint programming and
global optimization.

In particular, the interval Newton method for f(x)=0 works correctly
and optimally using forward operations in the evaluation of the
derivative and reverse multiplication in place of the final division.

1.4. Non-arithmetic operations related to intervals

A number of non-arithmetic operations are needed to support
the use of intervals. The list given in Part 5 is fairly liberal,
and in particular contains
- the tightest enclosure of a scalar product of two vectors of
  B-numerals (Section 5.1) to support applications to least
  significant bit algorithms; cf. Section 6.7;
- interval enclosures of bounds (Section 5.3) to support improved
  range enclosures based on monotonicity arguments;
- division with gap (Section 5.7) to support box splitting techniques
  in branch-and-bound methods;
- optional linear interpolation and extrapolation operations
  (Section 5.8), useful for applications in computational geometry.

Only the minimal support needed to enable these uses is provided.

1.5. Conversion of text and numerals

The conversion of text (in Part 6) and numerals (in Part 4) is
guided by the principle that it should allow rigor to be easily
specified by the user, and that conversion results in intervals

that preserve rigor throughout semantically correct programs.

In particular, it is considered to be semantically incorrect
- to use the constructor interval(x) with a numeral
  x not known to be finite and exact;
  instead one should use intervalabs or intervalrel; see Section 4.5;
- to use in a compiled arithmetic expression that results in an
  interval any inexact unary or binary arithmetic operation between
  two nonintervals;
- to use a binary operation involving an interval and a noninterval
  number if the latter is not known to be finite and exact;
- to introduce intervals except through operations conforming to the
  standard. (Additional care must be taken with intervals constructed
  from computed endpoints, and with operations producing nonstandard
  intervals; this may lead to semantic errors unless backed up by
  appropriate theory.)

For example, if zz is an interval variable, writing
   interval('1.1'), interval('1e309'), '1.1'*zz, or '1e309'*zz
or - in languages where user-defined literals are available -
   interval(1.1x), interval(1e309x), 1.1x*zz, or 1e309x*zz
is semantically correct. On the other hand, writing any of
   interval(1.1), interval(1e309), 1.1*zz, or 1e309*zz
is semantically incorrect, unless it is known
- either that 1.1 or 1e309 are represented exactly by B-numerals,
- or that these constants are translated at compile time to one of
  the correct forms mentioned above.

On the other hand, relying on the behavior of the compiler or the
underlying system is not recommended since it makes the program not
portable.

It is recommended that in introductions to interval computations,
beginners be made aware of the most frequent sources of semantic
errors:
- use of constructs like 0.1*xx or realHull(0.3,0.4), which does not
  account for conversion errors;
- use of f(x) in place of f(number2interval(x)) in overloaded function
  calls or arithmetic expressions, which does not account for rounding
  erros in the evaluation of f(x);
- use of constructs like f(number2interval(sup(xx))) for unbounded
  intervals xx (cf. Remark 2 in Section 4.1).

1.6. Numerals representing bounds

B-numerals (representing the bounds of intervals) are numerals
of a particular, distinguished format, most likely binary 64-bit
floating point data.

As stated, this excludes the consideration of multiple types for
interval arithmetic implementations. (However, changes to incorporate
multiple interval formats are minor, and could be accommodated at a
later stage.)

High precision rigorous calculations and calculations with intervals
with an extremely large bound are probably best done by representing
uncertain numbers as triples (x,ex,ee) consisting of a variable
precision (or dynamical precision) numeral x with the same basis
(or a power of it) as used for B-numerals, an integer exponent ex,
and a standard interval ee, representing an arbitrary number xtilde
with
   (xtilde-x)/basis^ex in ee.
This allows one to make optimal use of standard multiprecision packages
and standard interval arithmetic implementations.

An interval arithmetic package may also contain two or more different
implementations of the standard, the others being based upon the first
but using a different set of B-numerals. In this case, there should
be additional conversion routines between different types of B-numerals
such that conversion creates the smallest enclosure representable in
the output type.

To get enclosures for exact expressions of a prespecified accuracy,
(e.g., in order to implement Section 6.7),
one may compute the result first with normal intervals, then check
how much the resulting width is bigger than the desired width, and
increase the precision of the multiprecision part by this much
accuracy and a bit more for safety. If necessary (which is rarely the
case), this can be iterated. Thus no special variable precision
interval data type should be needed.

1.7. Relation to IEEE 754-2008

Care has been taken to make the bulk of the proposed standard
(Parts 1-6) independent of the arithmetical properties of numerals,
and hence of the IEEE 754-2008 floating-point standard.

However, Part 7 indicates desirable properties of directed rounding
deviating from the IEEE 754-2008 floating-point standard
that would facilitate the implementation of interval arithmetic
according to the specifications of this proposal.

Special attention is called to the static and dynamic rounding
attributes of IEEE 754-2008 clauses 4.1 and 4.2, in particular to
modes attached to significant blocks of program code, not just
single operations. Since interval computations are frequently
accompanied by scalar computations with directed rounding
(see the accompanying paper on ''Improving interval enclosures''),
proper support for such attributes is much more important than for
floating-point computations in general.

1.8. Levels of operations

A proposed minimal useful set of required operations are the following
operations defining core interval arithmetic.
- forward and reverse interval operations for
  plus, minus, times, divide, sqr (Sections 3.8-3.11 and 7.9)
- mixed operations with one interval argument for
  plus, minus, times, divide (Section 4.2)
- forward sqrt, exp, log (Sections 3.8-3.9)
- mid, rad (Sections 5.2 and 7.11)
- division with gaps (Section 5.7)
- optimal enclosure of sums and inner products (Section 5.1)

These operations might be singled out to represent (together with the
operations from Section 2.5) level 1 of the standard; the remaining
operations would represent level 2.
An appendix may suggest explicit model algorithms for all level 1
operations.

The standard might distinguish between standard-conforming
implementations that satisfy all requirements of the standard,
and standard-compatible implementations that satisfy the
requirements of the standard for all level 1 operations and for

those level 2 operations that are implemented, but that do not implement all level 2 operations. A standard-compatible implementation would list all implemented operations from the standard description.

1.9. Suggestions for hardware implementation

Level 1 operations implemented in hardware would significantly speed up existing applications in constraint programming, global optimization, and computational geometry.

For a hardware implementation, it must be taken into account that efficiency in applications of interval analysis depends on an effective interplay between ordinary (round to nearest) floating-point computations, interval computations (with outward rounding), and directed floating-point computations (in rounding modes up or down). See the section on directed rounding and hardware support in the accompanying document ''Improving interval enclosures''.

1.10. Optional features

In a number of cases of minor importance (often indicated by the word ''may'' or ''optional''), suggestions were made what might or might not be included into the standard.

Later sections contain also a number of nonnormative comments that explain certain details; these are indicated by listing them under the heading 'Remark(s)'.

1.11. Value-changing optimizations

These should be handled similar to Section 10.4 of IEEE 754-2008.

Transformations are allowed if and only if they would be exact in exact real arithmetic and provably lead to enclosures contained in the enclosures obtained by using the original expression. A (nonexhaustive) list of allowed and forbidden transformations should be provided in an appendix to the standard.

For example, zz = xx^2 or zz = xx*xx or the sequence of assignments

```
yy = xx; zz = xx*yy;
```

may be optimized to zz = sqr(xx), but [-2,3]*[-2,3] may not be
optimized to sqr([-2,3]).

Other example discussed: (x^2-y^2)/x^2 to 1-(x/y)^2.
What about xx*0 to 0? It changes the meaning if xx is Empty.

When interval arguments to a function are passed by value, the arguments
must be treated as if they were independent. When interval arguments
to a function are passed by reference, the arguments may be treated
by an optimizing compiler as the expression they denote.

The programmer must be able to enable or disable expression
transformations of the above kind for particular blocks of a program.
(Maybe require explicit enable to allow such optimizations?)

Should we allow replacing xx/yy by xx*inv(yy), which violates the
above requirement?

Some users want reproducibility. Some want speed. Some want the most
accurate results. How to cater for all?

(Unlike everything said before, the contents of this section is nowhere
detailed in the remainder of this proposal.)

1.12. Other issues

Things not yet treated which also need to be addressed:
- perhaps more usage suggestions/examples
- perhaps more implementation suggestions
- expand on Section 1.11
- consistent wording for requirements, recommendations, and options
  (similar to those in IEEE 754-2008)

Part 2. Representation and semantics

2.0. Theme of Part 2

This part defines the representation and semantics of intervals,
regarded as machine representations of uncertain real numbers.
The possible values are required to form a subset of the closed and
connected set of real numbers represented by the interval.

2.1. Assumptions about numerals

The representations of real numbers or pseudo-numbers (Inf, -Inf, NaN)
in the particular language in which the proposed standard
is made available (integers, floating-point numbers, fixed-point
numbers, etc.) are called L-numerals (L for language); there may be
several types of L-numerals, perhaps of different precision.

The representations of real numbers denoting bounds of intervals are
called B-numerals (B for bounds, see Section 1.6).
Numerals are either B-numerals or L-numerals.

If B-numerals are not among the L-numerals, they must be encodable
somehow in the language, and B-numeral and L-numeral must be convertible
into each other according to any of the rounding modes
roundTowardPositive, roundTowardNegative, roundTowardZero,
roundTiesToEven, and roundTiesToAway as specified in IEEE-754-2008.

The numerals representable in a particular implementation form a finite
set F. There is a partial mapping from F to R^* = R union {-Inf,+Inf},
the order completion of the set R of real numbers, which assigns to
certain elements x of F their value value(x).

NaN, -Inf, +Inf, and 0 denote any numeral x such that value(x) is
undefined, -Inf, +Inf, and 0, respectively; Inf and +Inf are used
synonymously. HUGEPOS and HUGENEG are B-numerals whose values are
closest to +Inf and -Inf, respectively.

B-numerals (representing the bounds of intervals) are numerals
of a particular, distinguished format, most likely binary 64-bit
floating point data.
(See Section 1.6 for possible multiple formats of B-numerals.)

It is required that NaN, -Inf, +Inf, 0, 1, and -1 exist both as
L-numerals and as B-numerals.

2.2. Representation of intervals, bounds, endpoints

An interval is represented by two B-numerals called the lower
bound and the upper bound. (But see Remark 2 of Section 2.3.)
Both the lower the and upper bound are referred to as a bound or
endpoint of the interval.

The representation is to be considered as a representation by types;
the specific format in which the lower and the upper bound are encoded
is left to the implementation; see Remark 2 in Section 2.3 for
useful alternative representations of lower or upper bounds, and
Section 7.0 for useful alternative representations of zero bounds.

In this document, intervals are written either with explicit bounds
as [l,u], with l and u replaced by the intended lower and upper bound,
respectively, or as single object by a repeated letter, such as xx,
yy, zz, aa, bb, cc, gg. In the latter case, a generic element of the
interval (i.e., the uncertain number represented by the interval)
is denoted by the corresponding single letter; e.g., x in xx.
Empty denotes the empty interval, and Entire the interval consisting
of all real numbers.

Two intervals are regarded as equal if the values of the lower bounds
agree and the values of the upper bounds agree, otherwise as distinct.
Which of the equivalent representations of an interval is used may
depend on the implementation.

Remark.
If there are B-numerals +0 and -0 both with value 0 then [-0,Inf] and
[+0,Inf] are equal. However, with the considerations of Part 7 in mind,
an implementation may choose to represent the interval [-0,Inf] always
as [+0,Inf]. Enforcing the latter form must then be taken care of by
the constructors.

2.3. Standard intervals

A standard interval has the form [l,u], where one of the following

holds:
  (i) Both bounds are finite, and l <= u.
 (ii) One bound is finite, and l = -Inf or u = +Inf.
(iii) l = -Inf and u = +Inf (entire interval Entire).
 (iv) l = NaN and u = NaN (empty interval Empty).
A standard interval [l,u] represents in case (i)-(iii) the set
(equivalently an arbitrary number from the set) of all real numbers x
with value(l) <= x <= value(u), and in case (iv) the empty set
(equivalently, no number).

In particular,
- standard intervals describe closed and connected sets of real
  numbers, including the empty set Empty = [NaN,NaN] and the set
  Entire = [-Inf,Inf] of all reals;
- numeral bounds with value 0, +Inf, -Inf, or NaN have the
  same meaning in standard intervals independent of their
  representation;
- bounds with subnormal numbers (formerly known as denormalized)
  are valid bounds for standard intervals;
- Since they are not real numbers, +Inf, -Inf, and NaN are not members
  of any standard interval (though they may be bounds);
- standard intervals do not represent closed disconnected sets such
  as [-Inf,-1] union [1,Inf].

Remarks.
  1. In IEEE 754-2008, there are multiple NaNs having a payload that
     propagates; see Section 7.2 of the IEEE 754-2008 standard.
     This may be used to implement labelled empty sets
     which carry debugging information about the operation which created
     the empty set in the first place, and thus to distinguish between
     empty sets produced by exceptions (''Not-an-Interval, NaI'') and
     genuine empty sets. (Note that such exceptions can only be due
     to interval constructors; interval arithmetic per se does not
     lead to exceptions.)

     The standard may perhaps describe how this feature should be used,
     and if so, how to query the label of an empty set, and how the
     labels should propagate in operations.
     This also needs a recommendation (to be added to Part 7)
     about how two NaNs with payload propagate under directed rounding
     in a binary operation (which is left open in IEEE 754-2008) since,
     if NaIs are supported, these must be propagated with higher
     priority than other payloads.

2. If B-numerals are symmetric around 0, the interval denoted in
   this document as [l,u] may be represented either as the pair (l,u),
   or as the pair(-l,u), or as the pair (l,-u), depending on the
   implementation.
   Thus it is permitted to implement consistently either the lower
   bound l by -l or the upper bound u by -u, in order to avoid
   frequent switches of the rounding mode; cf.
   http://kathrin.dagstuhl.de/files/Materials/06/06021/06021.LambovBranimir.Slides.pdf
   Recommended is in this case to invert the sign of l.

2.4. Nonstandard intervals

An interval [l,u] is nonstandard if it is not a standard interval.
No specification is made for the meaning of nonstandard intervals.

In particular, [-Inf,-Inf] and [+Inf,+Inf] are nonstandard intervals;
intervals with exactly one bound NaN, and intervals [l,u] with l>u
are also nonstandard.

2.5. Constructors and operations checking the semantics

Here for brevity 1 = true, 0 = false.

1. There is an operation Empty() with value Empty.
   (In programming languages, it is recommended to have a
   corresponding constant Empty.)
   There is an operation isEmpty(xx) that outputs 1 or 0
   depending on whether or not the interval xx is Empty.

2. There is an operation Entire() with value Entire.
   (In programming languages, it is recommended to have a
   corresponding constant Entire.)
   There is an operation isEntire(xx) that outputs 1 or 0 depending
   on whether or not the interval xx is Entire.

3. There is an operation isStandard(xx) that outputs 1 or 0
   depending on whether or not the input interval xx is standard.

4. There is an operation standard(xx) that outputs xx if xx
   is standard, and Empty otherwise.

5. There is an operation standardInterval(l,u) that returns the tightest interval containing the values of the two numerals l and u if l<=u, and Empty otherwise.

   l and u may be numerals of different type, in which case type overloading is needed.

6. There is an operation anyInterval(l,u) that creates the interval [l,u] from two B-numerals l and u without checking whether or not it is standard.

7. There is an operation areIdentical(xx,yy) that outputs for two intervals xx=[l,u] and yy=[l',u'] the value 1 if both l and l' represent the same value and u and u' represent the same value, or if all four bounds are NaN, and otherwise the value 0.

   There is an operation areDistinct(xx,yy) that outputs not(areIdentical(xx,yy)).

   Note the difference between areIdentical, areDistinct and the comparison operations equalToAA, unequalToAA from Section 5.4.

8. There is an operation isCompact(xx) that outputs 1 or 0 depending on whether or not the (standard or nonstandard) interval xx represents a compact textbook interval.

   Thus, the output is 1 for Empty and for standard intervals with finite bounds, and 0 for unbounded intervals and for nonstandard intervals.

9. There is an operation inf(xx) that returns for any interval xx=[l,u] the largest L-numeral of a given type smaller than or equal to l.

10. There is an operation sup(xx) that returns for any interval xx=[l,u] the smallest L-numeral of a given type larger than or equal to u.

11. There is an operation isIn(x,xx) or an operation contains(xx,x) (Which one of the two, or both?) that outputs 1 or 0 depending on whether or not the value of the numeral x is in the standard interval xx.

The behavior for a nonstandard interval xx is not specified.

In particular, isIn(x,xx) returns 0 when x is +-Inf or NaN and the interval is standard.

12. There is an operation dual(xx) that returns for any (standard or nonstandard) interval xx=[l,u] the (standard or nonstandard) interval [u,l].

Remarks.

1. If NaI (see Remark 1 of Section 2.3) is representable, there should also be operations that generate it and that test for it.

2. Perhaps one may add further operations isPositivelyBounded, isNegativelyBounded, isSingleton, although these can been easily constructed from the operations provided in Section 5.2.

2.6. Scope of operations

All operations with interval input shall be defined by the implementation both for standard and for nonstandard intervals, although the behavior for nonstandard input is not defined by this standard.

All operations with interval output permit both standard and nonstandard intervals as output. However, all arithmetic operations (see Part 3) have standard intervals as output if all interval inputs are standard intervals.

With the exception of the operations listed in Section 2.5, the standard makes no specification for the result of operations involving some nonstandard interval since there are several mutually conflicting extensions of the interval calculus interpreting [l,u] for l>u, including
- Kahan arithmetic (with nonstandard intervals interpreted as complements of open intervals),
- Kaucher arithmetic (where nonstandar intervals are ideal objects that allow one to define the inverse operation for addition),
- modal arithmetic (where standard and nonstandard intervals are interpreted as intervals in existential and univeral mode, respectively).

Though Kaucher arithmetic and modal arithmetic agree in many respects, they differ on noncommutable binary operations such as atan2.

2.7. Convention on using the word ''interval''

In the remainder of this document, all unqualified intervals will be assumed to be standard intervals (rather than nonstandard intervals or intervals in the mathematical sense).

Part 3. Arithmetic operations

3.0. Theme of Part 3

This part specifies the behavior for interval arguments of arithmetic operations either required by the standard, or of further implementation-dependent arithmetic operations. An implementation is standard-conforming only if all its required and all its implementation-dependent arithmetic operations conform to the requirements below.

3.1. Arithmetic operations on intervals

Arithmetic operations on intervals are based on corresponding real operations, whose names are used in overloading arithmetic expressions.

All interval operations are specified in such a way that the evaluation of an arithmetic expression involving intervals only contain all possible results of the arithmetic expression obtained by specializing each interval to an arbitrary real number contained in the interval.

In certain cases, the containment is pessimistic, in which case more elaborate range enclosure methods discussed in the literature must be used in place of simple interval evaluation.

3.2. Accuracy modes

Interval arithmetic operations are implemented in one of three accuracy modes, 'tightest', 'accurate', or 'valid'. An operation may be implemented in any accuracy mode if only 'valid' is required, in the accuracy modes 'tightest' or 'accurate' if 'accurate' is required, and must be implemented in the accuracy mode 'tightest' if 'tightest' is required.

There is a function accuracy(f) that outputs, for each arithmetic constant, unary operation, or binary operation with name passed as string f, the actually implemented accuracy mode (which may be tighter than required), or 'missing' if the string does not represent an implemented operation. This information should also be available at compile-time.

Remarks.

1. Note that modes are fixed in an implementation. There is no way for the user to change the accuracy mode of an arithmetic operation. There is only a way to lookup the mode and then to make (or not to make) use of it knowing how accurate it is.
2. If ENUM (enumerated) types are available, the output would naturally be ENUM; e.g., in C:

```
enum accuracy_enum { tightest, accurate, valid, missing };
typedef enum accuracy_enum accuracy_t;
```

3. A maximally portable program which wants to avoid the risk of producing nonidentical results on different implementations may not make use of any operation not in tightest mode.

3.3. Propagation of Empty

Independent of the accuracy mode, all arithmetic operations with one or more arguments Empty must have the result Empty, propagating labels in case of labelled empty sets. (The case of conflicting labels must be discussed.)

(Non-arithmetic operations need not propagate Empty.)

3.4. Test for continuity and undefined

For every unary operation <phi> and every binary operation <circ> which is not everywhere continuous, when calling the forward evaluation <phi>Hull (Section 3.9) or <circ>Hull (Section 3.11), respectively:

- The possiblyUndefined flag must be raised when the operands contain values for which the operation is not defined (not a finite real number), and
- The definedButPossiblyDiscontinuous flag must be raised when the operation is defined (a finite real number) for all values contained in the operands but is discontinuous for at least one of these values.

Examples.

(i) 1/xx with 0 in xx raises possiblyUndefined.
(ii) sqrt(xx) with xx=[-1,1] raises possiblyUndefined.
(iii) tan(xx) with xx=[1,2] raises possiblyUndefined even though the position of the pole at pi/2 may not be representable

as a B-numeral.
(iv) sign(xx) with xx=[-1,1] raises definedButPossiblyDiscontinuous.
(v) atan2(xx,yy) with xx=yy=[-1,1] raises definedButPossiblyDiscontinuous.

Remarks.
1. These flags are necessary in order to correctly infer validity and continuity of function evaluation, which is indispensable for the application of existence theorems.
2. Except in existence tests, the flags will probably never be inspected.

3.5. Usage of the word ''hull''

The hull of a set of real numbers refers to an interval (in the sense of this standard) containing the set, with the additional property that:
1. in accuracy mode 'tightest', the interval is the tightest interval containing the set.
2. in accuracy mode 'accurate', the interval is contained in the tightest interval containing all sets obtained from arguments whose Hausdorff distance from the inputs is at most one ulp of the corresponding inputs (in the sense that the symmetric difference consists at most of endpoints of the wider interval).
3. in accuracy mode 'valid', no further requirement is imposed.

3.6. Table: Arithmetic constants

pi	4*atan(1)
twopi	8*atan(1)
pihalf	2*atan(1)
e	exp(1)
goldenratio	(1+sqrt(5))/2,

and implementation-dependent arithmetic constants.

Remark.
These are only examples; the list may easily be modified.

3.7. Arithmetic constants (nullary operations) as intervals

For each arithmetic constant from Table 3.6,
there is a nullary operation resulting in the tightest interval
containing this constant.

In exact expressions (Section 6.7), nullary operations may be called
either by writing the name or by adding after the name the
characters (); e.g., both pi and pi() are acceptable.

3.8. Table: Unary arithmetic operations

The first column gives the name of a real unary operation, the second
its use in arithmetic expressions (Section 6.6), the third the minimal
accuracy requirement (Section 3.3) for the interval version as
specified in Section 3.9, the fourth whether a reverse mode
(Section 3.9) is required.

negation	- x	tightest	
square	sqr(x), x^2	tightest	reverse
inverse	inv(x), 1/x	tightest	reverse
abs	abs(x)	tightest	reverse
sign3	sign(x)	tightest	
ceiling	ceil(x)	tightest	
floor	floor(x)	tightest	
sqrt	sqrt(x)	tightest	
exp	exp(x)	tightest	
log	log(x), ln(x)	tightest	
log10	log10(x)	tightest	
log2	log2(x)	tightest	
intpow	x^k (see 3.12)	accurate	reverse
root	x^(1/q) (see 3.13)	accurate	reverse
ratpow	x^(p/q) (see 3.13)	accurate	reverse
sin	sin(x)	accurate	reverse
cos	cos(x)	accurate	reverse
tan	tan(x)	accurate	reverse
sinh	sinh(x)	accurate	
cosh	cosh(x)	accurate	reverse
tanh	tanh(x)	accurate	
asin	asin(x), arcsin(x)	accurate	
acos	acos(x), arccos(x)	accurate	
atan	atan(x), arctan(x)	accurate	
asinh	asinh(x), arsinh(x)	accurate	

```
     acosh           acosh(x), arcosh(x)        accurate
     atanh           atanh(x), artanh(x)        accurate
and implementation-dependent unary arithmetic operations with
implementation-specified calling syntax in arithmetic expressions
and arbitrary accuracy mode.

Here sign3 is the 3-valued sign, with sign(0)=0.

Remarks.
  1. The precise list of operations to require or to recommend may also
     be discussed; cf. Section 3.13 for root and ratpow.
  2. Suggested further unary operations (forward mode only; relevant for
     constructing optimal linear and quadratic underestimators):
        exp2(x)  := (e^x-1-x)/x^2
        sin2(x)  := (sin(x)-x)/x^2
        cos2(x)  := (cos(x)-1)/x^2
        sinh2(x) := (sinh(x)-x)/x^2
        cosh2(x) := (cosh(x)-1)/x^2
     extended by continuity to x=0.
  3. Other operations like
        expm1(x)  := e^x-1,
        exp1(x)   := (e^x-1-x)/x
        ratsin(x) := 2x/(1+x^2)
        ratcos(x) := (1-x^2)/(1+x^2)
     and endlessly many others may help in special cases...


3.9. Unary arithmetic operations on intervals

For each (partial) unary operation <phi> from Table 3.8,
there is a unary forward interval operation <phi>Hull defined by
   <phi>Hull(xx) = convex hull of the set of <phi>(x) with x in xx
                   such that <phi>(x) is defined and finite,
and, if required,
a binary reverse interval operation <phi>Inv defined by
   <phi>Inv(zz,xx) = convex hull of the set of x in xx for which
                     <phi>(x) is defined, finite, and in zz.
Here the second argument is optional, with default xx=Entire if absent.

Within arithmetic expressions, the unary operator <phi>
applied to an interval is always overloaded by <phi>Hull.

Note that inverse functions available as standard functions are
```

unary operations in their own right, with the usual definition.
Thus there will be a sqrthull in addition to sqrinv, and
    sqrthull(9)=3,   while   sqrinv(9)=[-3,3].

3.10. Table: Binary arithmetic operations

Notation is as in Section 3.8.

plus	x + y		tightest	
minus	x - y		tightest	
times	x * y		tightest	reverse
divide	x / y		tightest	reverse
min	min(x,y)		tightest	
max	max(x,y)		tightest	
pow	x^y	(see 3.12)	accurate	reverse
atan2	atan2(x,y)		accurate	reverse

and implementation-dependent binary arithmetic operations with
implementation-specified calling syntax in arithmetic expressions
and arbitrary accuracy mode.

Note that max and min are treated here as arithmetic operations, hence
return Empty when an argument is Empty, in apparent contrast to maxNum
and minNum in IEEE 754-2008, where NaNs are ignored. But this contrast
is spurious, as the semantics of NaN and Empty are different.

3.11. Binary arithmetic operations on intervals

For each (partial) binary operation <circ> from Table 3.10, there
should be a binary forward interval operation <circ>Hull defined by
   <circ>Hull(xx,yy) = xx <circ>Hull yy
                     = convex hull of the set of x <circ> y
                       with x in xx, y in yy, such that
                       x <circ> y is defined and finite.
and, if required,
two ternary reverse interval operations <circ>Inv1 and <circ>Inv2
defined by
   <circ>Inv1(bb,cc,xx) = convex hull of the set of x in xx
                          for which b in bb exists such that
                          x <circ> b is defined, finite, and in cc,
   <circ>Inv2(aa,cc,xx) = convex hull of the set of x in xx
                          for which a in aa exists such that
                          a <circ> x is defined, finite, and in cc.

Here the third argument is optional, with default xx=Entire if absent.

Within arithmetic expressions, the binary operator <circ>
applied to an interval is always overloaded by <circ>Hull.

If <circ> is commutative then <circ>Inv1 and <circ>Inv2 agree and may be
implemented simply as <circ>Inv.

Remark.
There is no need to spell out in the standard the mathematical
proposition which translates the above specification into tables with
formulas for the end points. These formulas, if desired, can be put
into an appendix, for all functions where someone is willing to
provide explicit formulas. It would be more useful to provide sample
implementations of a subset of the standard.

3.12. The power operation

The binary operation pow,
   where a pow b is defined for nonnegative a and real b,
   provided that b is positive when a is zero
must be complemented by a binary operation intpow,
   where a intpow p is defined for real a and integer p,
   provided that p is nonnegative when a is zero.

Remark.
In particular, 0 pow 0 is undefined, while 0 intpow 0 = 1.
This allows the polynomial f(x) = sum_{p=0:n} a_p x^p
to have the value p(0)=a_0 at x=0.

The binary operation intpow is viewed as a family of unary operations
parameterized by the exponent; thus the exponent is not
allowed to be an interval, intpowinv2 does not exist, and
intpowinv = intpowinv1 although the operation is not commutative.

For example,
   [0,1] intpowhull -2 = [0,Inf]   and   [0,1] powhull -2 = [0,Inf];
but
   [-1,1] intpowhull 3 = [-1,1]   while   [-1,1] powhull 3 = [0,1];
   0 intpowhull 0 = 1   while   0 powhull 0 = Empty.

3.13. Remarks

1. The above guarantees that the result of arithmetic operations
   with standard intervals is again a standard interval.

2. Overloading arithmetic expressions for interval arguments is always
   done by using the forward hull; in particular this applies to
   the evaluation referred to in Section 6.7.

3. Should all above operations be required, or should there be levels
   of conformance to the standard, with level 1 only catering for the
   most basic functionality?

4. There may also be a ternary operation ratpow(xx,p,q) implementing
   x^(p/q) for x>=0, integer p, and positive integer q, with x>0
   if p<0, viewed as a family of unary operations parameterized
   by p and q.
   Or a binary operation root implementing x^(1/q) for x>0 and a
   positive integer q, viewed as a family of unary operations
   parameterized by q.

5. There may also be a ternary operation ratmul(xx,p,q) for
   intervals xx and integers p, q with q>0, enclosing x*p/q for all
   x in xx, again viewed as a family of unary operations parameterized
   by p and q.
   This might be useful for interfaces with computer algebra packages.

Part 4: Mixed operations and type conversion

4.0. Theme of Part 4

This part specifies the conversion of numbers to intervals and the behavior of mixed arithmetic operations with an interval and a noninterval argument. These are important since they are often much faster than full interval operations.

Warning:

If the user means by a numeral anything different from what its value actually is, the user is required to pass the exact definition of the intended uncertain number as text (see Part 6) or via the constructors from Section 4.5.
Otherwise results obtained could be invalid (i.e. not honor guaranteed enclosure rules).

4.1. Conversion of numerals

There is an operation number2interval(x) that converts a numeral x to the tightest interval containing x if x is finite, but to Empty if x is infinite or NaN. In the latter case the nonstandardNumber flag is raised.

This ensures that
```
   xx = number2interval(x)  implies  isIn(x,xx)=not(isEmpty(xx)).
```

Remarks:

1. A similar but not identical effect as with xx = number2interval(x) is obtained with xx = realHull(x,x); cf. Section 5.1.
   The difference is that +-Inf are converted by the second statement to a nonempty set.

2. Care must be taken when converting intended infinities.

   For example, if fun(x) denotes a user-defined expression in the variable x that is monotone as a function of x then
   ```
       ff=convexHull(fun(iinf(xx)),fun(isup(xx)))
   ```
   (where iinf, isup, and convexHull are defined in Sections 5.3 and

5.6) yields an enclosure ff for fun(x) for all x in xx, which for compact xx is usually excellent, since in exact arithmetic the exact range is obtained. (For noncompact xx, the enclosure is less useful: E.g., both for fun(x)=(x-1)/(x+1), xx=[0,Inf], where the range is [-1,1], and for fun(x)=x^2-x, xx=[1,Inf], where the range is [0,Inf], we get Entire.)

However, writing instead

```
ff1=fun(number2interval(inf(xx)))
ff2=fun(number2interval(sup(xx)))
ff=convexHull(ff1,ff2)
```

gives an invalid result when xx has one or two infinite bounds. Indeed, a semantic error occurs since the infinite bound is treated as if it were a real number.
This problem is indicated by setting the nonstandardNumber flag. (The semantic error would probably be caught easily on debugging even without the flag, since instead of a wide result something very narrow is returned.)

3. I expect that the nonstandardNumber flag will never be inspected, except for debugging purposes. Since Empty will usually propagate very fast through complex calculations, this is the cheapest way to handle the exceptions.
(See also the remark in Section 2.3 on labelled versions of Empty, which would make setting the flag superfluous.)

4.2. Implicit conversion

Binary arithmetic operations from Table 3.10 with an interval argument and a numeral or text argument should give the same result as if a prior conversion of the noninterval according to Section 6.1 or 4.1 had been performed. This implicit conversion may be carried out outside or inside interval arithmetic proper.

Example.
For addition xx+y, where, for simplicity, xx=[l,u] and y are finite:
(i) external conversion: implementation as xx+interval(y);
(ii) internal conversion: implementation as [l+y,u+y].
The result is exactly the same, but (ii) can be cheaper
(not for + used here for simplicity, but for * and /)
and to be preferred.

4.3. Mixed binary operations

For each unary arithmetic operation from Section 3.8,
there is a unary operation that takes as input numerals and returns as
output the tightest enclosure of the exact result of the operation
if it is a real number, and Empty otherwise.

For each binary arithmetic operation from Section 3.10,
there is a binary operation that takes as input numerals and returns as
output the tightest enclosure of the exact result of the operation
if it is a real number, and Empty otherwise.

4.4. Conversion of numerals with known uncertainty

There is an operation intervalAbs(x,d) that returns for two
finite numerals x and d the tightest interval containing
all real numbers z with |z-x|<=d, but Empty if x is not finite
or d is NaN or -Inf, and Entire if d is +Inf.

There is an operation intervalRel(x,d) that returns for two
finite numerals x and d the tightest interval containing
all real numbers z with |z-x|<=d|x|, but Empty if x is not finite
or d is NaN or -Inf, and Entire if d is +Inf.

Remark.
Note that, except for x=0, intervalRel(x,TINY), where TINY denotes
the smallest positive numeral, has a similar effect as
inflate(number2interval(x)); cf. Section 5.3.

Part 5: Non-arithmetic operations

5.0. Theme of Part 5

This part defines non-arithmetic operations whose input or output involve intervals.

5.1. Interval-valued operations on noninterval arguments

1. There is an operation realHull(l,u) that returns for two numerals l and u the interval Empty if l or u is NaN, and otherwise the tightest interval containing
   l' = min(l,u,HUGEPOS) and u' = max(l,u,HUGENEG).
   (In the first case, the bounds may be required to preserve the payload of the NaN.)

2. There is a function sumAll(v) returning the tightest interval enclosing the sum of entries of a vector v of finite numerals of any fixed type, or Empty if one of the components is not finite.

3. There is a function innerProduct(v,w) returning the tightest interval enclosing a scalar product of two vectors v and w of finite numerals of any fixed type, or Empty if one of the components is not finite.

5.2. Numeral-valued unary operations on an interval

1. Midpoint and radius.
   There are an operation mid(xx) that returns for any interval xx a B-numeral m called the midpoint of xx, an operation rad(xx) that returns for any interval xx a B-numeral r called the radius of xx, and an operation midRad(xx) that returns both m and r.

   If xx=[l,u] is compact, m and r shall have the values defined by
   set round up
   r = 0.5*(u-l);m=l+r.
   If xx is Empty, m = NaN, r = NaN. (This happens automatically with the above algorithm.)
   If xx has an infinite bound, r = Inf, and m is the absolutely smallest number in xx,

```
        m = l,  r = Inf    if xx = [l,Inf]  with l>=0,
        m = u,  r = Inf    if xx = [-Inf,u] with u<=0,
        m = 0,  r = Inf    otherwise.

    Note that this guarantees the two essential properties
        mid(x) in xx       if xx is nonempty,
        |x-mid(x)|<=rad(x) for all x in xx.

2. Diameter, width.
   There is an operation diam(xx) or width(xx) that returns for any
   interval xx=[l,u] its diameter or width, the smallest B-numeral
   greater or equal to u-l, but NaN if xx is Empty.

3. Magnitude and mignitude.
   There is an operation mag(xx) that returns for any nonempty
   interval xx=[l,u] its magnitude, the smallest B-numeral greater
   or equal to max{abs(x) | x in xx}.
   There is an operation mig(xx) that returns for any nonempty
   interval xx=[l,u] its mignitude, the largest B-numeral less or
   equal to min{abs(x) | x in xx}.
   If xx is Empty, the B-numeral NaN is returned in both cases.

   Note that absHull(xx) = [mig(xx),mag(xx)].

4. Smallest.
   There is an operation smallest(xx) that returns for any nonempty
   interval xx the absolutely smallest B-numeral in the interval,
   i.e., 0 if xx contains 0, and the absolutely smaller bound
   otherwise. If xx is Empty, the B-numeral NaN is returned.

5. Zerolength.
   There is an operation zerolength(xx) that returns width(abs(xx)),
   i.e., max(-l,u) if 0 in xx=[l,u], and width(xx) otherwise.

6. Sign.
   There is an operation sign2(xx) that returns for any interval
   xx=[l,u] the 2-valued sign, namely
        -1 if 0>l<=u<=0,  +1 if 0<=l<=u>0,  NaN otherwise.

Remark.
If B-numerals are not L-numerals, the user may need to convert the
output of some of the above operations to L-numerals using the
conversion routines mentioned in Section 2.1, with an appropriate
```

rounding mode.

5.3. Interval-valued unary operations on an interval

1. There is an operation inflate(xx) that returns for any interval xx
   an interval [l,u] containing xx such that
   - the only numerals whose values are possibly contained in the set
     complement [l,u]\xx are l and u,
   - l<inf(xx) unless inf(xx)=-Inf,
   - u>sup(xx) unless sup(xx)=+Inf,

   Remarks.
   1. The IEEE 754-2008 functions nextUp and nextDown are likely
      to be used for the implementation.
   2. Should one provide eps-inflation for eps>0? Perhaps with an
      absolute and a relative version analogous to intervalAbs and
      intervalRel in Section 4.4 (which is a degenerate case of
      eps-inflation)?

2. There is an operation iinf(xx) that returns for any interval xx
   the tightest interval yy such that inf(yy)=inf(xx).

   Thus yy=[l,l] unless l=-Inf, in which case yy=[-Inf,HUGENEG].

3. There is an operation isup(xx) that returns for any interval xx
   the tightest interval yy such that sup(yy)=sup(xx).

   Thus yy=[u,u] unless u=Inf, in which case yy=[HUGEPOS,Inf].

   Remark.
   iinf and isup are useful for exploiting monotonicity
   (cf. Remark 2 in Section 4.1).

5.4. Boolean-valued binary operations on intervals

In this section, 1 or 0 stand for the Boolean values true and false.

1. Comparisons and disjointness.
   For each relation <omega> from the list
   - equalTo ==
   - unequalTo !=

lessThan <
greaterThan >
lessEqual <=
greaterEqual >=

there is an operation <omega>AA(xx,yy) that assigns to
any two intervals xx and yy the value 1 or 0 depending on whether
or not x <omega> y holds for all x in xx and all y in yy.

Remarks.

1. not(<omega>AA(xx,yy)) is generally not the same as
   <notomega>AA(xx,yy). Also, equalToAA and unequalToAA have
   a different meaning from isIdentical and isDistinct from
   Section 2.5.

2. unequalToAA(xx,yy) has the value 1 or 0 depending
   on whether or not xx and yy are disjoint, i.e., have no common
   element. This operation may also be provided in the additional
   form areDisjoint(xx,yy).

3. equalToAA(xx,yy) is 1 only if xx and yy are identical
   singleton intervals.

2. Variants of comparisons.
   There may be optional operations <omega>SS(xx,yy) that assign to
   any two intervals xx and yy the value 1 or 0 depending on whether
   or not x <omega> y holds for some x in xx and some y in yy.

   There may be optional operations <omega>AS(xx,yy) that assign to
   any two intervals xx and yy the value 1 or 0 depending on whether
   or not x <omega> y holds for all x in xx and some y in yy.

   There may be optional operations <omega>SA(xx,yy) that assign to
   any two intervals xx and yy the value 1 or 0 depending on whether
   or not x <omega> y holds for some x in xx and all y in yy.

   Remarks.
   1. In my opinion, the variants are rarely needed in applications,
      hence are taken as optional only.

   2. The AA mode and SS mode correspond to ''certainly'' and
      ''possibly'' versions of comparison operations in some
      existing implementations of interval arithmetic.

3. Containment.
   There is an operation containedIn(xx,yy) that assigns to
   any two intervals xx and yy the value 1 or 0 depending on whether
   or not x in yy for all x in xx.

   There is an operation containedInInterior(xx,yy) that assigns to
   any two intervals xx and yy the value 1 or 0 depending on whether
   or not x in yy for all x in xx, and x is not the value of a bound
   of yy.

   Remarks.
   1. According to standard topology, the empty set is (as here)
      contained in the interior of any set.

   2. There may be an optional operation containedInProperly(xx,yy)
      that assigns to any two intervals xx and yy the value 1 or 0
      depending on whether or not x in yy for all x in xx, and xx
      and yy are distinct.

5.5. Numeral-valued binary operations on intervals

1. Signed distance from interval.
   There is an operation signedDistance(x,yy) that returns for any
   numeral x and any interval yy=[l,u] as result
   - the B-numeral 0 if x in xx,
   - the largest B-numeral less or equal than x-l if x<l,
   - the smallest B-numeral greater or equal than u-x if x>u,
   - the B-numeral NaN otherwise, i.e., when x is NaN or xx is Empty.

2. Hausdorff distance.
   There is an operation distance(xx,yy) that returns for any two
   intervals xx and yy the smallest B-numeral greater
   or equal to their Hausdorff distance if xx and yy are nonempty,
   and otherwise the B-numeral NaN. (The Hausdorff distance is
   well-defined for nonempty sets only as a nonnegative extended
   real number.)

   See Section 7.10 for an explicit formula.

   If B-numerals are symmetric around 0 and x=[xl,xu], y=[yl,yu] then
      distance(xx,yy) = max(dly,duy,dxl,dxu),

where
dly=abs(signedDistance(xl,yy)),
duy=abs(signedDistance(xu,yy)),
dxl=abs(signedDistance(xx,yl)),
dxu=abs(signedDistance(xx,yu)).

3. Inner distance.
There is an operation innerDistance(xx,yy) that returns for any two intervals xx and yy the largest B-numeral smaller than or equal to the minimum of all |x-y| with x in xx, y in yy if xx and yy are nonempy, and the B-numeral NaN otherwise.

4. Projection to interval.
There is an operation projectToInterval(x,xx) that returns for any numeral x and any interval xx the finite numeral in xx closest to x and of the same type as x if such a numeral exists, and NaN otherwise.

Remarks.
1. This ensures that unless y:=projectToInterval(x,xx) is NaN, projectToInterval(x,xx) in xx.

2. If xx is not Empty and if x is finite and has the type of B-numerals, x is returned if x is in xx, and the bound of xx closest to x if x is not in xx.

3. If the type of x is different from that of B-numerals and x is not in xx, the closest bound must be rounded inwards and to a finite value, and the result must be checked for being still in xx; otherwise it is replaced by NaN.

4. It must be decided by the standard what to do if the type of x does not support NaN. Alternatively, one may restrict the use of projectToInterval to x with types supporting NaN.

Remark.
If B-numerals are not L-numerals, the user may need to convert the output of some of the above operations to L-numerals using the conversion routines mentioned in Section 2.1, with an appropriate rounding mode.

5.6. Interval-valued binary operations on intervals

1. Intersection.
   There is an operation intersection(xx,yy) that returns for any two intervals xx and yy the interval representing the intersection of xx and yy.

   Warning:

   Semantically invalid conversions (see Section 1.5) may result in spurious empty intersections, such as in case of
   ```
   xx=number2interval(0.1)*10
   yy=[0,1]
   zz=intersection(xx,yy)
   ```
   where zz is empty when binary 64-bit floating point data are used as B-numerals.

2. Convex hull.
   There is an operation convexHull(xx,yy) that returns for any two intervals xx=[l,u] and yy=[l',u'] the tightest interval containing the union of xx and yy.

   Remark.
   Unlike in arithmetic operations, an argument Empty does not propagate unless the other argument is Empty, too. Perhaps a flag should be raised to detect this behavior, at least if labelled Empty (see Remark 1 in Section 2.3) is used.

3. Inner operations.
   There is an operation innerAddition(xx,yy) that returns for any two intervals xx=[l,u] and yy=[l',u'] the tightest interval containing l+u' and u+l' if l+u'<=u+l', and Empty otherwise.

   There is an operation innerSubtraction(xx,yy) that returns for any two intervals xx=[l,u] and yy=[l',u'] the tightest interval containing l-l' and u-u' if l+u'>=u+l', and Empty otherwise.

5.7. Division with gap

There is an operation divisionWithGap(xx,yy) that returns for any two intervals xx=[l,u] and yy=[l',u'] the interval zz=divideHull(xx,yy) (see Section 3.11) and the gap gg, which is

- Empty if xx contains 0 or yy does not contain 0,
- Entire if xx or yy is Empty, and
- otherwise the widest subinterval whose interior cannot be
  realized as quotient x/y with x in xx, y in yy.

Remarks.

1. In a program, division with gaps must always be called as a function, whereas ordinary division (divideHull) will usually be called by overloading.
2. Similarly, one may provide perhaps implementations of inverse with gap, atan2 with gap, and inverse trigonometric or hyperbolic functions with gaps.

5.8. Linear interpolation

1. There may be an operation linearInt(xx,yy,tt) that provides, for any two compact intervals xx=[xl,xu], yy=[yl,yu], and any interval tt=[tl,tu] contained in [0,1] an enclosure of the set of all (1-t)*x+t*y such that x in xx, y in yy, t in tt, which is equivalent to or tighter than the result [l,u] of the following algorithm.

   ```
   set round down
   dl = yl-xl;tl1=(tl if dl>=0 else tu);l=xl+tl1*dl;
   set round up
   du = yu-xu;tu1=(tu if du>=0 else tl);u=xu+tu1*du;
   ```

   The behavior for noncompact xx or yy and for tt not in [0,1] is not specified by this proposal, but should be specified by each implementation.

2. There may be an operation linearExt(xx,yy,tt) that provides, for any two compact intervals xx=[xl,xu], yy=[yl,yu], and any interval tt=[tl,tu] contained in [1,Inf] an enclosure of the set of all (1-t)*x+t*y such that x in xx, y in yy, t in tt, which is equivalent to or tighter than the result [l,u] of the following algorithm.

   ```
   set round down
   dl = yl-xu;tl1=(tl if dl>=0 else tu);l=xu+tl1*dl;
   set round up
   du = yu-xl;tu1=(tu if du>=0 else tl);u=xl+tu1*du;
   ```

   The behavior for noncompact xx or yy and for tt not in [1,Inf] is not specified by this proposal, but should be specified by each implementation.

3. There may be an operation linearInv(xx,yy,zz,tt) that
   provides, for any four compact intervals xx, yy, zz, and
   tt=[tl,tu] with tl>=0 an enclosure of the set of all t in tt such
   that z=(1-t)*x+t*y for some x in xx, y in yy, z in zz, which is
   equal to or tighter than
      convexHull(timesInv(zz-xl,yy-xl,tt),timesInv(zz-xu,yy-xu,tt))
   with convexHull and timesInv defined in Sections 5.6 and 3.11,
   respectively.
   The behavior for noncompact xx, yy, or zz is not specified by
   this proposal, but should be specified by each implementation.

Remarks.

1. The extrapolation behavior for t in [-Inf,0] can be obtained by
   replacing t with 1-t. Cases where tt is an interval containing
   0 or 1 in the interior are not needed in computational
   geometry; not requiring them saves case distinctions.

2. Note that the four equations
      z=(1-t)*x+t*y,   y=(1-s)*x+s*z,
      x=(1-p)*y+p*z=(1-q)*z+q*y,   t=(z-x)/(y-x)
   are equivalent if s=1/t, p=1/(1-t), q=1/(1-s) and these quotients
   are defined, as are the conditions
      0<t<1, s>1, p>1, q<0
   and
      t>1, 0<s<1, p<0, q>0
   This explains the sign restrictions used.

5.9. Type conversion in non-arithmetic operations

All non-arithmetic operations shall allow for implicit type
conversion when a numeral appears in place of an interval
argument.

Remark.
This may need exceptions for certain cases where all interval
arguments are replaced by numerals and the operation already has a
different meaning for real arguments.

Part 6. Text to interval conversion

6.0. Theme of Part 6

This part defines which texts denoting a mathematical definition of an exact or uncertain real number shall be convertible into a standard interval guaranteed to contain any valid interpretation of this number.

This section is a discussion version only, without a full syntax specification; instead, it proceeds by examples.

6.1. General conversion rule

There is an operation text2interval(t) which returns for a given text t an interval containing a valid enclosure of all real numbers matching the definition given in the text according to the rules given in Sections 6.2-6.7 and raises a flag nonstandardNumber if the text cannot be interpreted by these rules or the interpretation results in a nonstandard interval. It may also raise additional implementation-dependent flags.

6.2. Textbook intervals

A text t taking one of the forms
    '[<literal_float>,<literal_float>]'
where <literal_float> stands for a string acceptable as input to the IEEE 754-2008 string->float conversions (clause 5.12 in 754-2008; this includes representations of Inf, -Inf, and NaN) followed and/or preceded by optional blanks, is converted by text2interval(t) into a (standard or nonstandard) interval whose lower bound is the largest B-numeral less or equal to the exact value of the literal lower bound (or Inf, -Inf, NaN, if the literal lower bound represents one of these), and whose upper bound is the smallest B-numeral greater or equal to the exact value of the literal upper bound (or Inf, -Inf, NaN, if the literal upper bound represents one of these).

6.3. Centered intervals

A text t taking the form
    '<<literal_float>+-<nonneg>>',
where <literal_float> is as in Section 6.2, <nonneg> is a nonnegative
<literal_float>, and the outer pointed brackets denote these as
characters, is considered to be an interval, and is required to be
converted by text2interval(t) into the tightest interval containing the
numbers <literal_float>-<nonneg> and <literal_float>+<nonneg>.
If <literal_float> is not finite, Empty is returned.

Remark.
I prefer the notation <2.3+-0.005> to the Intlab style centered interval
<2.3,0.005> since it is suggestive of its meaning and more easily
readable. Note that 2.3+-0.005 without pointed brackets would be
ambiguous since it is equivalent to the exact expression 2.3-0.005
(Section 2.6) if negation has higher precedence than plus.

6.4. Uncertain numbers

A text t taking one of the forms
    '<fixpart>?<expart>', '<fixpart>_<expart>',
    '<fixpart>?<unctype><fixpart><expart>',
where <fixpart> is a decimal <literal_float> without exponent part,
<expart> is the possibly empty exponent part of a decimal
<literal_float> with decimal exponent indicator 'e',
and <unctype> is one of the characters in 'udar'
(standing for ''up'', ''down'', ''absolute'', and ''relative'')
is considered to be an uncertain number, and is required to be
converted by text2interval(t) into the tightest interval containing
all real numbers represented by this uncertain number.

(The precise semantics must be described here.)

The use of 'd' as decimal exponent indicator (as is customary for
Fortran double precision numbers) would conflict with the use of 'd'
for representing downward uncertainty.

Examples.
(ulp is one unit in the last place of the decimal number displayed)
12.3_ # represents 12.3 +- 1 ulp, i.e., [12.2,12.4]
1.23?e3 # represents 1.23e3 +- 1/2 ulp, i.e., [12250,12350]

```
    Note: An earlier version had here erroneously 1 ulp
1.23?5e-1 # represents 1.23e-1 +- 5 ulp, i.e., [0.118,0.128]
.23?u2 # represents .23 + <=2 ulp, i.e., [.23,.25]
.23?d2 # represents .23 - <=2 ulp, i.e., [.21,.23]
.234?a0.01 # represents .234 +- 0.01, i.e., [.224,.244]
.23?r0.1e2 # represents .23e2 * (1+-0.1), i.e., [20.7,25.3]
```

6.5. Exact numbers

A text t taking one of the forms
'<sign><natural>/<natural>' or '<literal_float>',
where <sign> is empty or one of the characters + or -, <natural>
is a natural number in decimal notation without exponent,
and <literal_float> is as in Section 6.2, is considered to be an exact
number, and is required to be converted into the tightest interval
containing this number.

Remark.
Very long rationals frequently arise in problems generated
from computer algebra packages using rational arithmetic.
For example, I have seen rationals with 800 digits in a particular
constraint satisfaction problem posed to me, generated for a
computer-assisted proof in the geometry of numbers.

6.6. Use at compile-time

Conversion according to Section 6.1-6.5 should yield the same result
regardless of whether it is done at compile time or at run time.

Remark.
In the future C++0x standard, conversion according to Section 6.1-6.5
could be done via user defined literals
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2765.pdf
Thus exact numbers 2, 0.1, and 1e400 can be encoded as
2x, 0.1x, 1e400x, etc.
and intervals as
"[1.1,1.2]"x, ".23?r0.1e-5"x, etc.
(Previously, the suffix i was used in place of x, but i might be
used also for complex literals. Coordination with the ISO C++ committee
may be needed to decide on the best recommendation.)

6.7. Exact constant expressions

A text t taking the form of an arithmetic expression in the operations from Section 3.6, 3.7, 3.8, and 3.10 below (including standard-conforming operations which a particular implementation provides), containing no variables but only constants of the form '<literal_float>' or '<literal_float>x', is considered to be an exact constant expression.

It is recommended that there be a library routine exactExpression(t) that converts an exact expression into an interval containing this number, at least as tight as if the corresponding computation would have been done by converting each of the constants to intervals and then performing the corresponding interval operations.

Remarks.

1. Precedence must still be specified!

2. This facilitates the accurate enclosure of exact values of compound expressions such as
   goldenratio = (1+sqrt(5))/2,
   continuedfraction = 1/(1-1/(1-1/(1-1/(1-1/(1-1/123456789))))),
   pisixth = atan(1)*2/3,
   pisixth = arctan(1)*2/3,
   pisixth = pi()/6,
   pisixth = pi/6, etc.
   In particular (whether or not atan is in the mandatory list of unary operations), text2interval(t) should recognize both strings '4*atan(1)' and '4*arctan(1)' and get a valid enclosure for pi returned if the arc tangent is implemented.

3. The formulation above is such that conversion may be done either by calling the corresponding converters of '<literal_float>' or '<literal_float>x', followed by the interval operations, or in a more involved way that gives extra accuracy. A high quality implementation would probably do this using multiprecision arithmetic or a staggered correction format. See also Section 1.6.

6.8. Interval to text conversion

There is an operation text(xx,m) which returns for a given interval xx

```
and a given conversion mode m a text t interpretable by text2interval
such that
- the textbook interval equivalent to the text t contains xx,
- the only numerals whose values are possibly contained in the set
  complement text2interval(t)\xx are the endpoints of text2interval(t).
An implementation shall supply at least modes returning
- decimal textbook intervals as in Section 6.2,
- hexadecimal textbook intervals as in Section 6.2,
- uncertain numbers as in Section 6.4,
At least one ''exact'' conversion mode must return for every interval xx
a text t such that text2interval(t) recovers xx exactly.

Remark.
Inexact conversion modes (e.g., binary to decimal) are the rule;
in those cases one cannot always avoid losing some accuracy.
If the B-numerals are binary, the exact conversion mode would
most likely be hexadecimal conversion.
```

Part 7: Useful directed rounding properties

7.0. Theme of Part 7

This optional part describes desirable properties of floating-point operations under directed rounding. These facilitate the implementation of the proposed standard when all zero bounds are represented as +0 for lower bounds and -0 for upper bounds (rather than as an arbitrary zero), provided that the implementation of the constructors in Section 2.5 and Part 6 and of all operations resulting in intervals ensure this represetation of zeros.

In this part we assume that each B-numeral is represented as a floating-point datum according to the IEEE 754-2008 standard. Listed are desirable deviations from the standard behavior of operation; these deviations are restricted to the behavior in the two rounding modes
- up = roundTowardPositive and
- down = roundTowardNegative.

7.1. Behavior for zero values

All arithmetic operations (in the sense of IEEE 754-2008) with a result having the value 0 (whether exact or inexact) should represent the result as -0 in rounding mode up, and as +0 in rounding mode down.

In an interval context where lower bounds are rounded down and upper bounds are rounded up, this corresponds to returning the tightest bounds if one thinks of -0 (resp. +0) being infinitesimally smaller (resp. larger) than 0. For exact zeros this may disagree with the rules stated in clause 6.3 of IEEE 754-2008.

Remark.
These conventions on signed zeros need not be satisfied for quiet operations such as minNum and maxNum. (Do not confuse these with the interval operations min and max, overloaded as minHull and maxHull, defined in Section 3.10-3.11.)

7.2. Product zero times infinity

The product of a number with value 0 and a number with value
+-Inf should have the value 0, with sign as specified in Section 7.1.

7.3. Adding and subtracting infinities

The result of x-x for x=Inf or x=-Inf, and the result of
x+y for x=Inf, y=-Inf or x=-Inf, y=Inf should be 0, with sign as
specified in Section 7.1.

Remark.
Note that +-Inf are used as bounds only, not as numbers.

7.4. Division by +0

The result of the division of a number x by +0 should be
NaN if x is NaN, +Inf if x>0, -Inf if x<0, and if x=0 then 0, with
sign as specified in Section 7.1.

7.5. Division by -0

The result of the division of a number x by -0 should be
NaN if x is NaN, -Inf if x>0, +Inf if x<0, and if x=0 then 0, with
sign as specified in Section 7.1.

7.6. Quotient of infinities

The result of x/y when x and y are infinite should be the
value 1 if x and y have the same sign, and -1 otherwise.

This amounts to considering Inf as a fixed huge number, and -Inf
as its negative. Any finite values with the same signs would do
as well.

7.7. Disabling denormalized numbers

Disabling denormalized numbers is harmless for most applications

of interval arithmetic as long as in rounding mode up (down), the
computed result is greater (smaller) than or equal to the result
obtained in exact arithmetic.

7.8. Benefits of the deviant behavior

If these deviations from the IEEE 754-2008 standard are realized,
and bounds l=0 are represented by +0, bounds u=0 are represented by -0,
then the statements 7.9-7.11 hold:
(Someone please check this! I hope I didn't forget any case.)

7.9. Binary operations +,-,*,/

For <circ> in {+,-,*,/},
   [l,u] <circ> [l',u'] = [l'',u'']
with
   l'' = min(l <circ> l', l <circ> u', u <circ> l', u <circ> u')
         in rounding mode down,
   u'' = max(l <circ> l', l <circ> u', u <circ> l', u <circ> u')
         in rounding mode up
will give the correct results in all cases with two exceptions:

(i) For <circ> = / and l' < 0 < u', [l,u] not [0,0]
     we must have l''=-Inf, u''=Inf,
(ii) For <circ> = / and [l',u'] = [0,0],
     we must have l''=NaN, u''=NaN.

Remark.
Note that by Section 7.1, all zero arguments of the min (or max)
will have the same sign, so that the IEEE 754-2008 behavior of
minNum and maxNum will produce results conforming to Section 7.1.

7.10. Hausdorff distance

The formula
    set round up
    distance([l,u],[l',u']) = max(abs(l-l'),abs(u-u'))
for the Hausdorff distance (Section 5.5) works correctly for all
intervals.

7.11. Linear interpolation

The algorithms from Section 5.8 also work correctly
when xx or yy or both are noncompact.

7.12. Midpoint and radius

The proposed formulas for midpoint and radius in Section 5.2. need
the exceptions mentioned, no matter how one specifies the exceptional
behavior of the arithmetic operations under directed rounding.

References

[1] R.B. Kearfott, M.T. Nakao, A. Neumaier, S.M. Rump, S.P. Shary and P. van Hentenryck, Standardized notation in interval analysis, pp. 106-113 in: Proc. XIII Baikal International School-seminar "Optimization methods and their applications", Irkutsk, Baikal, July 2-8, 2005. Vol. 4 "Interval analysis". Irkutsk: Institute of Energy Systems SB RAS, 2005.
http://www.mat.univie.ac.at/~neum/papers.html#notation

[2] R.E. Moore, Methods and applications of interval analysis, SIAM, Philadelphia, 1979.

[3] A. Neumaier, Interval methods for systems of equations, Encyclopedia of Mathematics and its Applications 37, Cambridge Univ. Press, Cambridge 1990.

[4] A. Neumaier, Computer graphics, linear interpolation, and nonstandard intervals, Manuscript (2008).
http://www.mat.univie.ac.at/~neum/papers.html#nonstandard

[5] A. Neumaier, Improving interval enclosures, Manuscript (2008).
http://www.mat.univie.ac.at/~neum/papers.html#encl