P1788: IEEE Standard For Interval Arithmetic Version 05.1

John Pryce and Christian Keil, Technical Editors
To the P1788 reader.

General. Passages in this color are my editorial comments: mostly asking for answers to or debate on a question; or giving my opinion; or noting changes made.

Ignore text like this: (some text)→delete It belongs in the full text but isn’t relevant to the part currently under discussion, and it didn’t seem worth rewriting the whole sentence.

Definitions. Christian Keil has done a lot of work on these. We believe they are now reasonably complete, and consistent with each other and the text. Only those definitions relevant to Level 1 are included in the current text.

Required and recommended elementary point functions. The lists in §7.6, 7.7 have not been voted on in their present form. I formed those in §7.6.2, 7.7.1 from the original lists of Jürgen Wolff von Gudenberg in Motion 10, much modified by subsequent discussion.

Maybe the group will accept them as they are; but if they prove controversial, we shall need separate motions to discuss them.

Christian has taken on the task of coordinating changes to these lists: adding or deleting functions, or moving functions between Required and Recommended.

Provisional items. The subclauses §7.6.7 Constructors, and §7.6.8 Numeric functions of intervals, are provisional, and not part of the present motion. Constructors are the subject of Motion 30. I look forward to someone submitting a motion to decide §7.6.8.

Notes to version of 2012/02/12. Main changes from the previous version of 2011/12/05 are as follows.

(a) §7.6.9 Boolean functions of intervals. This is now part of the text to be voted on. I had forgotten we passed 2 motions on comparisons!

The 7 Kulisch comparisons are included as Required operations. So are isEmpty, isEntire, and areDisjoint, following the Vienna proposal.

(b) I have made the motion 21 “overlapping” function a Recommended operation (this is open to change) and given a brief explanation of how it can be used as a primitive for implementing other comparisons.

(c) I had also forgotten inner addition and subtraction (Motion 12). These are in §??? but not exactly as in Motion 12. See my justification there.

(d) It seems to me that Motion 9, on complete arithmetic and exact dot product, is entirely about finite precision issues. In Level 1 it is now not mentioned except for (in the Required Operations) a forward reference to Level 2. Also, because of the difference in intrinsic complexity, I think it is correct not to list dot product among the basic arithmetic operations.

(e) A new subclause “Constants” is inserted at the start of §7.6 “Required operations”. Ian McIntosh made me realize something on this is needed.

(f) Text strings are explicitly stated (§7.1) to be things one can talk about at Level 1.

(g) In the numeric functions of intervals, “diameter” renamed “width” as being shorter, and the function renamed from diam to wid.

(h) The I, etc., notation has been slightly altered so that the I can be regarded as an operator on sets of numbers. This is explained in §3.1

\footnote{1CK}
CHAPTER 1

Introduction
1. Overview

⚠ Scope and Purpose taken word for word from the Project Authorization Request (PAR), but changed from future to present tense. Other IEEE front-matter, such as list of participants, to be included in due course.

1.1. Scope. This standard specifies basic interval arithmetic (IA) operations selecting and following one of the commonly used mathematical interval models and at least one floating-point type defined by the IEEE-754/2008 standard. Exception conditions are defined and standard handling of these conditions are specified. Consistency with the model is tempered with practical considerations based on input from representatives of vendors and owners of existing systems.

The standard provides a layer between the hardware and the programming language levels. It does not mandate that any operations be implemented in hardware. It does not define any realization of the basic operations as functions in a programming language.

1.2. Purpose. The aim of the standard is to improve the availability of reliable computing in modern hardware and software environments by defining the basic building blocks needed for performing interval arithmetic. There are presently many systems for interval arithmetic in use, and lack of a standard inhibits development, portability, and ability to verify correctness of codes.

1.3. Inclusions. This standard specifies
- Types for interval data based on underlying floating-point formats.
- Constructors for intervals from floating-point and character sequence data.
- Addition, subtraction, multiplication, division, fused multiply-add, square root, and other interval-valued operations for intervals.
- Midpoint, radius and other numeric functions of intervals.
- Interval comparison relations.
- Required elementary functions.
- Data type and operations for the calculation of exact sums or dot-products.
- Conversions between different interval types.
- Conversions between interval types and external representations as character sequences.
- Interval-related exceptions and their handling.

1.4. Exclusions. This standard does not specify
- Which floating-point formats supported by the underlying system shall have an associated interval type.
- Details of how an implementation represents intervals internally by floating-point numbers. However, interchange types for intervals are specified.

⚠ To be revised later.

1.5. Word usage.

In this standard three words are used to differentiate between different levels of requirements and optionality, as follows:
- may indicates a course of action permissible within the limits of the standard with no implied preference (“may” means “is permitted to”);
- shall indicates mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (“shall” means “is required to”);
- should indicates that among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (“should” means “is recommended to”).

Further:
- might indicates the possibility of a situation that could occur, with no implication of the likelihood of that situation (“might” means “could possibly”);
- see followed by a number is a cross-reference to the clause or subclause of this standard identified by that number;
The verb *comprise* is used to indicate that members of a set are exactly those objects having some property, e.g. “the set of mathematical intervals comprises the closed, connected subsets of \( \mathbb{R} \)”; an unqualified *consist of* merely asserts all members of a set have some property, e.g. “a binary floating point type consists of numbers with a terminating binary representation”. “Comprises” can usually be replaced by “consists exactly of”.

*Note* and *Example* introduce text that is informative (that is, is not a requirement of this standard).

1.6. The meaning of conformance. This standard specifies interval arithmetic in terms of abstract interval types. These need not be, but probably will be, described by floating point numbers of the underlying floating point system. An implementation, to conform to this standard, shall support at least one interval type. It probably will support several such types. The standard does not require the underlying floating point system to be 754-conforming. However, it introduces the notion of a 754-conforming implementation, having more stringent requirements of accuracy, and of support for mixed-type interval operations.

1.7. Programming environment considerations.

This standard does not define all aspects of a conforming programming environment. Such behavior should be defined by a programming language definition supporting this standard, if available, and otherwise by a particular implementation. Some programming language specifications might permit some behaviors to be defined by the implementation.

**Language-defined behavior** should be defined by a programming language standard supporting this standard. Then all implementations conforming both to this interval standard and to that language standard behave identically with respect to such language-defined behaviors. Standards for languages intended to reproduce results exactly on all platforms are expected to specify behavior more tightly than do standards for languages intended to maximize performance on every platform. Because this standard requires facilities that are not currently available in common programming languages, the standards for such languages might not be able to fully conform to this standard if they are no longer being revised. If the language can be extended by a function library or class or package to provide a conforming environment, then that extension should define all the language-defined behaviors that would normally be defined by a language standard.

**Implementation-defined behavior** is defined by a specific implementation of a specific programming environment conforming to this standard. Implementations define behaviors not specified by this standard nor by any relevant programming language standard or programming language extension. Conformance to this standard is a property of a specific implementation of a specific programming environment, rather than of a language specification. However a language standard could also be said to conform to this standard if it were constructed so that every conforming implementation of that language also conformed automatically to this standard.
2. Ideas underlying the standard

This introduction explains some of the alternative interpretations, and sometimes competing objectives, that influenced the design of this standard, but is not part of the standard.

2.1. Mathematical context. Interval computation is a collaboration between human programmer and machine infrastructure which, correctly done, produces mathematically proven numerical results about continuous problems—for instance, rigorous bounds on the global minimum of a function or the solution of a differential equation. It is part of the discipline of “constructive real analysis”. In the long term, the results of such computations may become sufficiently trusted to be accepted as contributing to legal decisions. The machine infrastructure acts as a body of theorems on which the correctness of an interval algorithm relies, so it must be made as reliable as is practical. In its logical chain are many links—hardware, underlying floating-point system, etc.—over which this standard has no control. The standard aims to strengthen one specific link, by defining interval objects and operations that are theoretically well-founded and practical to implement.

Of various forms of interval arithmetic (IA) the most widely used is set-based, where an interval is a set of reals: it has established software to find validated solutions of linear and nonlinear algebraic equations, optimization problems, differential equations, etc. It was agreed early on that this standard should extend the “classical” arithmetic on bounded, closed, nonempty real intervals as used by R.A. Moore [3], by including unbounded closed real intervals and the empty set.

A different form is Kaucher IA, where an interval is formally just a pair \((a, b)\) of real numbers, which for \(a \leq b\) is “proper” and identified with the normal interval \([a, b]\), and for \(a > b\) is “improper”—or the closely related modal arithmetic. It is commercially important in that it supports the currently fastest implementations of validated interpolation and related algorithms for high-end graphics rendering used in the film industry.

In view of their significance the working group wished to support both forms. Because of their different mathematical foundations this led to the concept of flavors. A flavor is a version of IA that extends classical IA in a precisely defined sense, such that when only classical intervals and restricted operations are used (avoiding for instance division by an interval containing zero), exactly the same results are produced independent of flavor, both at the mathematical level and in finite precision.

Currently the standard incorporates two flavors, set-based and Kaucher; others may possibly be added in the future, since several other extensions of classical IA exist.

To minimize complexity, the standard for each flavor is presented as a separate sub-document within the overall standard, readable as a self-contained unit without reference to other flavors, and with common clauses that specify how a flavor extends classical IA.

The set-based flavor is presented first, on the grounds that it is the more accessible, being relatively easy to grasp, easy to teach, and easy to interpret in the context of real-world applications. In this theory:

- Intervals are sets.
- They are subsets of the set \(\mathbb{R}\) of real numbers—precisely, all closed and connected (in the topological sense) subsets of \(\mathbb{R}\).
- An interval operation is defined algebraically, in contrast to topologically. The interval version of an elementary function such as \(\sin x\) is essentially the natural extension to sets of the corresponding pointwise function on real numbers.

This contrasts on the one hand with Kaucher theory, and on the other hand with containment set (cset) theory, where intervals are subsets of the extended reals \(\overline{\mathbb{R}}\), and operations are defined topologically, in terms of limits.

2.2. Specification Levels. The 754-2008 standard describes itself as layered into four Specification Levels. To manage complexity, P1788 uses a corresponding structure. It deals mainly with Level 1, of mathematical interval theory, and Level 2, the finite set of interval datums in terms of which finite-precision interval computation is defined. It has some concern with Level 3, of representations of intervals as data structures; and none with Level 4, of bit strings and memory.
There is another important player: the programming language. We acknowledge the experience of the 754-2008 working group, who recognized a serious defect of the 754-1985 standard, namely that it specified individual operations but not how they should be used in expressions. Over the years, compilers made clever transformations so that it became impossible to know the precisions used and the roundings performed while evaluating an expression, or whether the compiler had even “optimized away” \((1.0 + x) - 1.0\) to become simply \(x\).

This is also a problem for intervals. Thus the standard makes requirements and recommendations on language implementations, thereby defining the notion of a standard-conforming implementation of intervals within a language.

The language does not constitute a fifth level in some linear sequence; from the user’s viewpoint it sits above datum level 2, alongside theory level 1, as a practical means to implement interval algorithms by manipulating Level 2 entities (though most languages have influence on Levels 3 and 4 also).

### 2.3. The Fundamental Theorem

Moore’s Fundamental Theorem of Interval Arithmetic (FTIA) is central to interval computation. Roughly, it says as follows. Let \(f\) be an explicit arithmetic expression—that is, it is built from finitely many elementary functions (arithmetic operations) such as \(+, -, \times, \div, \sin, \exp, \ldots\), with no non-arithmetic operations such as intersection, so that it defines a real function \(f(x_1, \ldots, x_n)\). Then evaluating \(f\) “in interval mode” over any interval inputs \((x_1, \ldots, x_n)\) is guaranteed to give an enclosure of the range of \(f\) over those inputs.

A version of the FTIA holds in all variants of interval theory, but with varying hypotheses and conclusions. In the context of this standard, an expression should be evaluated entirely in one flavor, and inferences made strictly from that flavor’s FTIA; otherwise, a user may believe an FTIA holds in a case where it does not, with possibly serious effects in applications. As stated, the FTIA is about the mathematical level. Moore’s achievement was to see that “outward rounding” makes the FTIA hold also in finite precision, and to follow through the consequences. An advantage of the level structure used by the standard is that the mapping between levels 1 and 2 defines a framework where it is easily proved that the finite-precision FTIA holds in any conforming implementation.

Generally it can only be determined a posteriori whether the conditions for any version of the FTIA hold; this is an important application of the standard’s decoration system.

For each flavor in the standard, its subdocument must state precisely the form of the FTIA it obeys, both at the mathematical level 1 and at the finite-precision level 2.

### 2.4. Operations

There are several interpretations of evaluation outside an operation’s domain and operations as relations rather than functions. This includes classical alternative meanings of division by an interval containing zero, or square root of an interval containing negative values. To illustrate the different interpretations, consider \(y = \sqrt{x}\) where \(x = [-1, 4]\).

1. In optimization, when computing lower bounds on the objective function, it is generally appropriate to return the result \(y = [0, 2]\), and ignore the fact that \(\sqrt{\cdot}\) has been applied to negative elements of \(x\).

2. In applications where one must check the hypotheses of a fixed point theorem are satisfied (such as solving differential equations):
   (a) one may need to be sure that the function is defined and continuous on the input and, hence, throw an illegal argument exception when, as in the above case, this fails; or
   (b) one may need the result \(y = [0, 2]\), but must flag the fact that \(\sqrt{\cdot}\) has been evaluated at points where it is undefined or not continuous.

3. In constraint propagation, the equation is often to be interpreted as: find all \(y\) such that \(y^2 = x\) for some \(x \in [-1, 4]\). In this case the answer is \([-2, 2]\).

The standard provides means to meet these diverse needs, while aiming to preserve clarity and efficiency.

In the context of flavors, a key idea is that of common operation instances: those elementary interval calculations that at the mathematical level are required to give the same result in all flavors. For example \([1, 2] / [3, 4] = [1/4, 2/3]\) is common, while division by an interval containing zero is not common.
2.5. Decorations.

Many interval algorithms are only valid if certain mathematical conditions are satisfied: for instance one may need to know that a function, defined by an expression, is (everywhere) defined and continuous on the box in \( \mathbb{R}^n \) defined by \( n \) input intervals \( x_1, \ldots, x_n \). The IEEE 754 model of global flags to record events such as division by zero was considered inadequate in an era of massively parallel processing. Thus, in this standard, such events are recorded locally.

A 1788 decorated interval is an ordinary interval tagged with a few bits (the decoration) that can record, e.g., “while evaluating this expression, each elementary function was defined and continuous on its inputs”—which implies the same for the function defined by the whole expression.

Care was taken to meet different user needs. Bare (undecorated) intervals are available for simple use without validity checks. Decorated intervals are recommended for serious programming, but suffer the “17-byte problem”: a typical bare interval stored as two doubles takes up 16 bytes, so a decorated one needs at least 17 bytes. On large problems this may cause great inefficiencies—in data throughput if one stores 17-byte data structures, or in storage if one pads the structure out to, say, 32 bytes. Hence a compressed decorated interval scheme is provided for advanced use. It aims to give the speed of 16-byte objects, at a cost in flexibility but supporting applications such as checking whether a function is defined and continuous on its inputs.
3. Notation, abbreviations, definitions

3.1. Frequently used notation and abbreviations.

- **R**: the set of real numbers.
- **R**: the set of extended real numbers, \( \mathbb{R} \cup \{ -\infty, +\infty \} \).
- **I\R**: the set of closed real intervals, including unbounded intervals and the empty set.
- **F, G, ...**: generic notation for (the set of numbers, including \( \pm \infty \)) representable in some floating point format.
- **IF, IG, ...**: the members of **I\R** whose lower and upper bounds are in **F, G, ...**.
- **Empty**: the empty set.
- **Entire**: the whole real line.
- **NaI**: Not an Interval.
- **NaN**: Not a Number.
- **qNaN**: quiet NaN.
- **sNaN**: signaling NaN.
- **x, y, ...**: generic notation for a numeric value [resp. numeric function].
- **f, g, ...**: generic notation for an interval value [resp. interval function].
- **f, g, ...**: generic notation for an expression, producing a function by evaluation.
- **Domain(f)**: the domain of a point-function \( f \).
- **Range(f | s)**: the range of a point-function \( f \) over a set \( s \); the same as the image of \( s \) under \( f \).

[Note. Little used in this document, but used in classical interval analysis, are **I\R**, the set of bounded, nonempty closed real intervals; and **IF**, the intervals of **I\R** whose bounds are in **F**. The symbols **I** and **I** act as operators on subsets \( S \) of **R**, namely **I** or **I**(\( S \)) = “the empty set, plus all intervals whose endpoints are in **S**” and **I** = “all nonempty intervals whose endpoints are finite and in **S**”.]

3.2. Definitions.

\[\Delta\] Definitions belonging to Levels 2 onward have been temporarily removed.

Dan Zuras notes that the Definitions subclause should be self-contained, i.e. one does not need to look outside it to understand a definition, at least in outline. Please check if this rule is flouted.

3.2.1. **arithmetic operation.** A function provided by an implementation (see Definition 3.2.8). It comes in three forms: the point operation, which is a mathematical real function of real variables such as \( \log(x) \); one or more interval versions, each being an interval extension of the point operation; and one or more decorated interval versions, each being a decorated interval extension of the point operation. Together with the interval non-arithmetic operations (§7.4.1), these form the implementation’s library, which splits into the point library containing point functions and the interval library containing interval functions.

A **basic arithmetic operation** is one of the six functions +, -, \times, \div, fused multiply-add **fma** and square root **sqrt**.

Constants such as 3.456 and \( \pi \) are regarded as arithmetic operations whose number of arguments is zero. Details in §7.4.4.

3.2.2. **box.** See Definition 3.2.10.

3.2.3. **domain.** For a function with arguments taken from some set, the domain comprises those points in the set at which the function has a value. The domain of an arithmetic operation is part of its definition. E.g., the (point) arithmetic operation of division \( x/y \), in this standard, has arguments \( (x, y) \in \mathbb{R}^2 \), and its domain is the set \( \{ (x, y) \in \mathbb{R}^2 \mid y \neq 0 \} \). See also Definition 3.2.15.

3.2.4. **elementary function.** Synonymous with arithmetic operation.

3.2.5. **expression.** A symbolic object \( f \) that is either a symbolic variable or, recursively, of the form \( \phi(g_1, \ldots, g_k) \), where \( \phi \) is the name of a \( k \)-argument arithmetic or non-arithmetic operation and the \( g_i \) are expressions. It is an arithmetic expression if all its operations are arithmetic.

\[\text{CK 2011-10-15 Not in the motion.}\]
operations. Writing \( f(z_1, \ldots, z_n) \) makes \( f \) a bound expression, giving it an argument list comprising the variables \( z_i \) in that order, which must include all those that occur in \( f \).

3.2.6. fma. Fused multiply-add operation, that computes \( x \times y + z \). One of the basic arithmetic operations.

3.2.7. hull. (Full name: interval hull.) When not qualified by the name of a finite-precision interval type, the hull of a subset \( s \) of \( \mathbb{R} \) is the tightest (q.v.) interval containing \( s \).

3.2.8. implementation. When used without qualification, means a realization of an interval arithmetic conforming to the specification of this standard.

3.2.9. inf-sup. Describes a representation of an interval based on its lower and upper bounds.

3.2.10. interval. A closed connected subset of \( \mathbb{R} \); may be empty, bounded or unbounded. May be called a 1-dimensional interval, see next paragraph. The set of all intervals is denoted \( \mathbb{IR} \).

A box or interval vector is an \( n \)-dimensional interval, i.e. a tuple \((x_1, \ldots, x_n)\) where the \( x_i \) are intervals. Often identified with the cartesian product \( x_1 \times \ldots \times x_n \subseteq \mathbb{R}^n \), it is empty if any of the \( x_i \) is empty. Details in 7.2.

3.2.11. interval extension. An interval extension of a point function \( f \) is a function \( f \) from intervals to intervals such that \( f(x) \) belongs to \( f(x) \) whenever \( x \) belongs to \( x \) and \( f(x) \) is defined. Details in 7.4.3.

3.2.12. interval function, interval mapping. A function from intervals to intervals is called an interval function if it is an interval extension of a point function, and an interval mapping otherwise. Details in 7.4.3.

3.2.13. interval library. See Definition 3.2.1.

3.2.14. interval version. See Definition 3.2.1.

3.2.15. natural domain. For an arithmetic expression \( f(z_1, \ldots, z_n) \), the natural domain is the set of \( x = (x_1, \ldots, x_n) \in \mathbb{R}^n \) where the expression defines a value for the associated point function \( f(x) \). See 7.7.

3.2.16. no value vs. undefined. Some functions do not have a defined value at the mathematical model of Level 1 (see Clause 4). This translates to a defined value at the interval datum Level 2 given at the corresponding places in the standard. Therefore the term “no value” is not to be confused with an “undefined” value, which has the common meaning of a value not determined by the standard and thus being free for the implementation to decide.

3.2.17. non-arithmetic operation. An operation on intervals that is not an interval extension of a point operation; includes interval intersection and union.

3.2.18. number. Any member of the set \( \mathbb{R} \cup \{-\infty, +\infty\} \) of extended reals: a finite number if it belongs to \( \mathbb{R} \), else an infinite number. See 7.9.

3.2.19. point function, point operation. A mathematical function of real variables: that is, a map \( f \) from its domain, which is a subset of \( \mathbb{R}^n \), to \( \mathbb{R}^m \), where \( n \geq 0, m > 0 \). It is scalar if \( m = 1 \). Any arithmetic expression \( f(z_1, \ldots, z_n) \) defines a (usually scalar) point function, whose domain is the natural domain of \( f \).

3.2.20. point library. See Definition 3.2.1.

3.2.21. range. The range, \( \text{Range}(f \mid s) \), of a point function \( f \) over a subset \( s \) of \( \mathbb{R}^n \) is the set of all values that \( f \) assumes at those points of \( s \) where it is defined, i.e. \{ \( f(x) \mid x \in s \text{ and } x \in \text{Domain } f \} \).

3.2.22. string. A text string, or just string, is a finite sequence of characters belonging to some alphabet. See 7.1.

3.2.23. tightest. Smallest in the partial order of set containment. The tightest set (unique, if it exists) with a given property is contained in every other set with that property.
4.1. Specification levels overview.

The standard is structured into four levels, summarized in Table 1 that match the levels defined in the 754 standard, see 754 Table 3.1.

Level 1, in Clause 7, defines the mathematical theory underlying the standard. The entities at this level are mathematical intervals and operations on them. Conforming implementations shall implement this theory.

(In addition to an ordinary (bare) interval, this level defines a decorated interval, comprising a bare interval and a decoration. Decorations implement the P1788 exception handling mechanism.)

Level 2, in Clause 8, is the central part of the standard. Here the mathematical theory is approximated by an implementation-defined finite set of entities and operations. A level 2 entity is called a datum (plural “datums” in this standard, since “data” is often misleading).

An interval datum is a mathematical interval tagged by a unique type T. The type abstracts a representation scheme—a particular way of representing intervals (e.g., by storing their lower and upper bounds as IEEE binary64 numbers). Level 2 arithmetic normally acts on intervals of a given type to produce an interval of the same type (but interval operations that act on intervals of types other than the result type are possible).

Level 3, in Clause 9, is concerned with the representation of interval datums—usually but not necessarily in terms of floating point values. A level 3 entity is an interval object. Representations of decorations, hence of decorated intervals, are also defined at this level.

The Level 3 requirements in this standard are few, and concern mappings from internal representations to external ones, such as interchange formats and I/O.

Level 4, in Clause 10, is concerned with the encoding of interval objects. A level 4 entity is a bit string. This standard makes no Level 4 requirements.

The arrows in Table 1 denote mappings between levels. The phrases in italics name these mappings. Each phrase “total, many-to-one”, etc., labeled with a letter a to d, is descriptive of the mapping and is equivalent to the corresponding labeled fact below.

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Mathematical Model level.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>Number system used by flavor F.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Principles of how +, −, ×, ÷ and other arithmetic operations are extended</td>
<td></td>
</tr>
<tr>
<td></td>
<td>to intervals.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>↓ T-interval hull</td>
<td>identity map ↑</td>
</tr>
<tr>
<td></td>
<td>total, many-to-one</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Level 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A finite subset T of the F-intervals—the T-interval datums—and operations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>on them.</td>
<td>Intermal datum level.</td>
</tr>
<tr>
<td></td>
<td>“represents” ↑</td>
<td></td>
</tr>
<tr>
<td></td>
<td>partial, many-to-one, onto</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Level 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Representations of T-intervals, e.g. by two floating point numbers.</td>
<td>Representation level.</td>
</tr>
<tr>
<td></td>
<td>“encodes” ↑</td>
<td></td>
</tr>
<tr>
<td></td>
<td>partial, many-to-one, onto</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Level 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Encodings 0111000...</td>
<td>Bit string level.</td>
</tr>
</tbody>
</table>

Table 1. Specification levels for interval arithmetic

4. Structure of the standard in levels
d. Not every interval encoding necessarily encodes an interval object, but when it does, that object is unique. Each interval object has at least one encoding and may have more than one.

5. Flavors

5.1. Flavors overview. The standard permits different interval flavors, which embody different foundational (Level 1) approaches to intervals. An implementation may support more than one flavor. Flavor is a property of program execution context, not of an individual interval, therefore just one flavor shall be in force at any point of execution. It is recommended that at the language level, the flavor should be constant at the level of a procedure/function, or of a compilation unit. The permitted flavors are described in this document.

The flavor concept enforces a common core of behavior that different kinds of interval arithmetic must share:

- The set of required operations, identified by their names, is the same in all flavors. Similarly the set of recommended operations is the same in all flavors. See §7.

⚠️ Is it OK not to tabulate that set in this introduction, but to do so in the text for each flavor, thus minimizing changes to the existing text of the set-based flavor?

- There is a set of common intervals whose members are—in a sense made precise below—intervals of any flavor.

- There is a set of common evaluations of library operations, with common intervals as input, that give—again in a sense made precise below—the same result in any flavor.

[Examples. The intervals $[−1, 4]$ and $[5, 6]$ are common. The evaluation $[−1, 4] + [5, 6]$ is common and shall give the result $[4, 10]$ in any flavor. The square root evaluation $\sqrt{[−1, 4]}$ is not common; its result is $[0, 2]$ in the set based flavor, and is undefined in the Kaucher flavor. The intersection evaluation $[−1, 4] \cap [5, 6]$ is not common; its result is Empty (not common) in the set based flavor, and is the improper interval $[5, 4]$ (not common) in the Kaucher flavor. The $\sqrt{[−1, 4]}$ example illustrates that the inputs of an evaluation, and its result in some flavor, can be common, while the evaluation itself is not common.]

5.2. Definition of common intervals and common evaluations. The choice of the set $\mathcal{C}$ of common intervals and the set $\mathcal{CE}(\phi)$ of common evaluations of an operation $\phi$ is a design decision that defines the flavor concept. It should aim for simplicity, and $\mathcal{CE}(\phi)$ should be specified by a general rule that makes it easy to add a new operation to the library if needed. The choice that was made is specified in the following paragraphs.

All likely flavors extend the classical Moore arithmetic \[3\] on the set $\mathbb{IR}$ of closed bounded nonempty real intervals, and no other intervals belong to all of them. Hence, the chosen set of common intervals is $\mathbb{IR}$.

The common evaluations are specified in terms of graphs of interval operations. For an interval operation $\phi$ of arity $k$, its graph (in some flavor) is a subset of a $(k+1)$-dimensional space of intervals, namely the set of interval $(k+1)$-tuples $(x_1, x_2, \ldots, x_k; y)$ such that $y = \phi(x_1, x_2, \ldots, x_k)$ is true in that flavor. Each such tuple is called an operation instance.

The general rule is that each $\phi$ has a set $\mathcal{CE}(\phi)$ of common evaluations: a set of operation instances $(x_1, x_2, \ldots, x_k; y)$ such that all its components are in $\mathbb{IR}$ and

$$y = \phi(x_1, x_2, \ldots, x_k) \quad \text{shall hold in all flavors,}$$

in the sense made precise below. It may be regarded as the common graph of $\phi$. An operation must be one-valued—i.e., a function—so if $(x_1, x_2, \ldots, x_k; y)$ and $(x_1, x_2, \ldots, x_k; z)$ are both in $\mathcal{CE}(\phi)$ then $y = z$. The specific choice of $\mathcal{CE}(\phi)$ is as follows.

Arithmetic operation: that is, an interval extension of the corresponding point function $\phi$. The common operation instances are those $(x_1, x_2, \ldots, x_k; y)$ such that the point function $\phi$ is everywhere defined and continuous on the closed, bounded, nonempty box $(x_1, x_2, \ldots, x_k)$, and $y$ equals the range of $\phi$ over this box. Then necessarily $y$ belongs to $\mathbb{IR}$.

Non-arithmetic operation: The common operation instances are those tuples with common inputs $x_i$ such that the result $y$ is also common. In particular for $\text{ConvexHull}$, the common operation instances are those $(x_1, x_2; y)$ with arbitrary $x_1, x_2 \in \mathbb{IR}$ and $y$ equal to the convex hull of $x_1 \cup x_2$. For intersection, they are those $(x_1, x_2; y)$ with...
arbitrary \( x_1, x_2 \in \mathbb{R} \) and \( y \) equal to \( x_1 \cap x_2 \), provided the latter is nonempty (since \( \emptyset \notin \mathbb{R} \)).

Other operations: \( \Delta \) (if any). To be written.

[Examples. From the previous example, the triple \([-1, 4]; [5, 6]; [4, 10]\) is a common operation instance for addition: it is in \( \mathcal{CE}(\{+\}) \).

The pair \([-1, 4]; [0, 2]\) is an operation instance of \( \sqrt{\cdot} \) in the set-based flavor. But in the Kaucher flavor, \([-1, 4]; y\) cannot be an operation instance for any \( y \), since \( \sqrt{-1, 4} \) is undefined. Thus it also cannot be common: there is no \( y \) for which it is in \( \mathcal{CE}(\{\sqrt{\cdot}\}) \).

Similarly, \([-1, 4]; [5, 6]; \emptyset\) is an operation instance of \( \cap \) in the set-based flavor, and \([-1, 4]; [5, 6]; \emptyset\) is an operation instance of \( \cap \) in the Kaucher flavor. Thus, there is no \( y \) for which the triple \([-1, 4]; [5, 6]; y\) is common and is in \( \mathcal{CE}(\{\cap\}) \).

5.3. Relation of common evaluations to flavors. Defining the meaning of common evaluations must take into account that the common intervals are not necessarily a subset of the intervals of a given flavor, but are identified with a subset of it by an embedding map.

[Examples. A Kaucher interval is defined to be a pair \((a, b)\) of real numbers—equivalently, a point in the plane \( \mathbb{R}^2 \)—which for \( a \leq b \) is “proper” and identified with the normal real interval \([a, b]\), and for \( a > b \) is “improper”. Thus the embedding map is \( x \mapsto (\inf x, \sup x) \) for \( x \in \mathbb{R} \).

For the set-based flavor, every common interval is actually an interval of that flavor \((\mathbb{R} \text{ is a subset of } \mathbb{R}^\text{F})\), so the embedding is the identity map \( x \mapsto x \) for \( x \in \mathbb{R} \).]

Formally, a flavor is identified by a pair \((\mathfrak{F}, f)\) where \( \mathfrak{F} \) is a set of Level 1 entities, the intervals of that flavor, and \( f \) is a one-to-one embedding map \( \mathbb{R} \rightarrow \mathfrak{F} \). Usually, \( f(x) \) is abbreviated to \( \mathfrak{F}x \).

It is then required that operation compatibility shall hold for each library operation \( \phi \) and for each flavor \((\mathfrak{F}, f)\). Namely, given \( x_1, x_2, \ldots, x_k \) and \( y \) in \( \mathbb{R} \),

If \((x_1, x_2, \ldots, x_k); y\) is a common operation instance of \( \phi \),

then \((\mathfrak{F}x_1, \mathfrak{F}x_2, \ldots, \mathfrak{F}x_k); \mathfrak{F}y\) is an operation instance of \( \phi \) in flavor \( \mathfrak{F} \).

That is, if \( \phi(x_1, x_2, \ldots, x_k) \) has the common value \( y \), then \( \phi(\mathfrak{F}x_1, \mathfrak{F}x_2, \ldots, \mathfrak{F}x_k) \) must be defined in \( \mathfrak{F} \) with value \( \mathfrak{F}y \).

An evaluation in \( \mathfrak{F} \) of a single operation, that has the form \([\mathfrak{F}\phi(x_1, x_2, \ldots, x_k); \mathfrak{F}y]\), is called a common evaluation of that operation. An evaluation in \( \mathfrak{F} \) of an expression, in which only common evaluations of operations occur, is called a common evaluation of that expression.

That is, a common evaluation of an expression in a flavor \((\mathfrak{F}, f)\) is one where the inputs are members of \( f(\mathbb{R}) \), and each intermediate value is produced by a common evaluation of an elementary operation so that it is also in \( f(\mathbb{R}) \); hence the final result is in \( f(\mathbb{R}) \).

The decoration system makes it possible to determine, for a specific expression and specific interval inputs, whether common evaluation has occurred, see \( \S 5.5 \).

5.4. Flavors and the Fundamental Theorem. At Level 1, common evaluation of an expression \( f \) gives the same result in all flavors, “modulo the embedding map”. That is, if, inputs \( x_1, \ldots, x_n \), all in \( \mathbb{R} \), are given to an expression, and the resulting evaluation is common and produces an output \( y \) in \( \mathbb{R} \), then evaluating the same expression in \( \mathfrak{F} \) with inputs \( x_j^* = \mathfrak{F}x_j \) gives an output \( y^* \) which equals \( \mathfrak{F}y \).

Suppose \( f \) is an arithmetic expression (does not contain set operations such as intersection) so that it defines a real point function \( f(x_1, \ldots, x_n) \). If the above evaluation is common then the classical FTIA \((2.3)\) can be stated in the following form: \( f \) is everywhere defined and continuous on the box \( x = (x_1, \ldots, x_n) \), and the output \( y \) encloses the range of \( f \) over \( x \).

This remains true, modulo the embedding map, in any flavor. That is, suppose inputs \( x_j^* \in \mathfrak{F} \) are given to the expression and evaluation gives output \( y_j^* \in \mathfrak{F} \). Suppose the inputs are common intervals and it is known, via the decoration system or otherwise, that the evaluation was common. Then one can map back to corresponding inputs \( x_j \in \mathbb{R} \) and \( y \in \mathbb{R} \), and draw the conclusions of the classical FTIA. As noted in \( \S 2.3 \) this holds also in finite precision.

This “minimal” FTIA derives automatically from the classical FTIA and the meaning of common evaluation.
§6.0

[Note. Each of the set-based and Kaucher flavors has a more general FTIA that contains the minimal FTIA as a special case. Some flavors have a generalized meaning of the “contains” relation, which is used in stating their FTIA: e.g., the Kaucher flavor defines \([a, b] \supseteq [c, d]\) to mean \((a \leq c \land b \geq d)\), whether \([a, b]\) and \([c, d]\) are proper or improper. However, since the minimal FTIA is defined by mapping back to common intervals, it does not need to use any such generalized notions.]

5.5. Flavors and the decoration system. △ This was not in Motion 36, but is my proposal for a “minimal” scheme.

To support the notion of flavor-independent execution, it needs to be possible for a program to determine that a specific evaluation of an expression was common, and moreover to do this in a flavor-independent way. This is done by the decoration system (Subclause 7.8).

This determination must be done at Level 2, because of the possibility of overflow, in the following generalized sense: an evaluation \(\phi(x_1, \ldots, x_k)\) of an arithmetic operation has inputs \(x_i\) that are common intervals, and its Level 1 result \(y^{(1)}\) is a common interval, but the implementation cannot find a Level 2 common interval \(y^{(2)}\) of the destination type \(T\), that contains \(y^{(1)}\). This is usually because a \(y^{(2)}\) does not exist, but possibly because it is very expensive to discover for certain whether it exists or not.

To give the needed information, there shall be a decoration \(\text{com}\) (meaning “common”) in all flavors. Each arithmetic operation \(\phi\) shall raise \(\text{com}\) (or a decoration that implies it), if: its input box \(x = (x_1, \ldots, x_k)\) consists of common intervals; and the implementation shows \(\phi\) is defined and continuous on \(x\), and finds a common interval \(y^{(2)}\) of the destination type that contains the Level 1 result \(y^{(1)}\).

The decoration mechanism (correctly initialized) ensures, in any flavor, that if the final output of an arithmetic expression evaluation is decorated \(\text{com}\), then that evaluation was indeed common so that the conditions of the minimal FTIA are satisfied.

[Examples. Reasons why an individual evaluation of \(\phi\) with common inputs \(x = (x_1, \ldots, x_k)\) may not raise \(\text{com}\) include the following.

1. The implementation finds \(\phi\) is not defined and continuous everywhere on \(x\). Examples: \(\sqrt{[-4, 4]}\), \(\text{sign}([0, 2])\).

2. The Level 1 result is common but cannot be enclosed in a suitable common Level 2 interval. Example: 
   \(2 \ast [0, \text{REALMAX}]\) in an inf-sup type.

3. It is too expensive to determine whether \(y^{(1)}\) can be enclosed in a suitable common Level 2 interval.
   Example: \(\cot([x, x])\) where \(x\) is a large number very close to a multiple of \(\pi\).
]

Flavors should define other decoration values, but \(\text{com}\) is the only one that is required to have the same meaning in all flavors.

6. Conformance requirements

△ Omitted because based on the draft of Sep. 2011, which was written for set-based flavor only.
CHAPTER 2

Set-Based Intervals

This Chapter contains the standard for the set-based interval flavor.

7. Level 1 description

In this clause, subclauses §7.1 to §7.5 describe the theory of mathematical intervals and interval functions that underlies this standard. The relation between expressions and the point or interval functions that they define is specified, since it is central to the Fundamental Theorem of Interval Arithmetic. Subclauses §7.6, 7.7 list the required and recommended arithmetic operations (also called elementary functions) with their mathematical specifications. (Subclause 7.8 describes, at a mathematical level, the system of decorations that is used among other things for exception handling in P1788.)

7.1. Non-interval Level 1 entities. In addition to intervals, the standard deals with entities of the following kinds. They may be used as inputs or outputs of operations.

- The set \( \mathbb{R} = \mathbb{R} \cup \{ -\infty, +\infty \} \) of extended reals. Following the terminology of 754 (e.g., 754\$2.1.25), any member of \( \mathbb{R} \) is called a number: it is a finite number if it belongs to \( \mathbb{R} \), else an infinite number.

  An interval’s members are finite numbers, but its bounds—if the interval is non empty—can be infinite. Finite or infinite numbers can be inputs to interval constructors, as well as outputs from operations, e.g., the interval width operation.

- The set of (text) strings, namely finite sequences of characters chosen from some alphabet. Since Level 1 is primarily for human communication, the standard makes no Level 1 restrictions on the alphabet used. Strings may be inputs to interval constructors, as well as inputs or outputs of read/write operations.

7.2. Intervals. The set of mathematical intervals supported by this standard is denoted \( I_\mathbb{R} \).

It comprises the empty set (denoted \( \emptyset \) or Empty) together with all nonempty closed intervals \( x \) of real numbers having the form

\[
x = [\underline{x}, \overline{x}] := \{ x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x} \},
\]

where \( \underline{x} \) and \( \overline{x} \), the bounds of the interval, are extended-real numbers satisfying \( \underline{x} \leq \overline{x} \leq +\infty \) and \( \underline{x} > -\infty \).

[Notes.]

- The above definition implies \( -\infty \) and \( +\infty \) can be bounds of an interval, but are never members of it. For instance, \( [1, +\infty] \) denotes in this document the interval \( \{ x \mid 1 \leq x < +\infty \} \).

- Mathematical literature generally uses a round bracket, or reversed square bracket, to show that an endpoint is excluded from an interval, e.g. \( (a, b) \) and \( [a, b) \) denote \( \{ x \mid a < x \leq b \} \). This notation is used when convenient, such as in the tables of domains and ranges of functions in §7.6, 7.7.

- The collection of intervals \( I_\mathbb{R} \) could be described more concisely as comprising all sets \( \{ x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x} \} \) for arbitrary extended-real \( \underline{x}, \overline{x} \).

  However, this obtains Empty in many ways, as \( [\underline{x}, \overline{x}] \) for any bounds satisfying \( \underline{x} \geq \overline{x} \), and also as \( [-\infty, -\infty] \) or \( [+\infty, +\infty] \). The description (3) was preferred as it makes a one-to-one mapping between valid pairs \( \underline{x}, \overline{x} \) of endpoints and the nonempty intervals they specify.

- The notation is chosen so that \( I \) is an operator: for any subset \( S \) of the extended reals \( \mathbb{R} \), \( I_S \) or \( I(S) \) comprises Empty together with all intervals (3) whose bounds belong to \( S \). This is used at Level 2 to specify finite-precision types, e.g. \( I(\text{binary64}) \) which comprises Empty together with all nonempty intervals whose bounds are IEEE754 double precision numbers.

\(^{1}\text{CK} \) Decorations are not part of the motion
Excluding Empty, Entire, and semi-infinite intervals is indicated by the notation \( \mathbb{I} \), e.g. \( \mathbb{I} \mathbb{R} \) or \( \mathbb{I}(\mathbb{R}) \), the nonempty closed intervals with (finite) real bounds, is the set used in classical interval literature such as the work of R.E. Moore.

\[ \mathbb{I} \text{ Revert to previous notation, remove I.} \]

A box or interval vector is an \( n \)-tuple \((x_1, \ldots, x_n)\) whose components \( x_i \) are intervals, that is, a member of \( \mathbb{IR}^n \). Usually \( x \) is identified with the cartesian product \( x_1 \times \cdots \times x_n \) of its components, a subset of \( \mathbb{R}^n \). In particular \( x \in \mathbb{R}^n \), for \( x \in \mathbb{R}^n \), means by definition \( x_i \in x_i \) for all \( i = 1, \ldots, n \); and \( x \) is empty if (and only if) any of its components \( x_i \) is empty.

7.3. Hull. The (interval) hull of an arbitrary subset \( s \) of \( \mathbb{R}^n \), written \( \text{hull}(s) \), is the tightest member of \( \mathbb{IR}^n \) that contains \( s \). (The tightest set with a given property is the intersection of all sets having that property, provided the intersection itself has this property.)

7.4. Functions.

7.4.1. Function terminology. In this standard, operations are written as named functions; in a specific implementation they might be represented by operators (i.e., using some form of infix notation), or by families of type-specific functions, or by operators or functions whose names might differ from those in this standard.

The terms operation, function and mapping are broadly synonymous. The following summarizes the usage in this standard, with references in parentheses to precise definitions of terms.

- A point function (\$7.4.2\$) is a mathematical real function of real variables. Otherwise, function is usually used with its general mathematical meaning.
- A (point) arithmetic operation (\$7.4.2\$) is a mathematical real function for which an implementation provides versions in the implementation’s library (\$7.4.3\$).
- A version of a point function \( f \) means a function derived from \( f \), such as an interval extension (\$7.4.3\$) of it; usually applied to library functions.
- An interval arithmetic operation is an interval version of a point arithmetic operation (\$7.4.3\$).
- An interval non-arithmetic operation is an interval-to-interval library function that is not an interval arithmetic operation (\$7.4.3\$).
- A constructor is a function that creates an interval from non-interval data (\$8.11.7\$).

7.4.2. Point functions. A point function is a possibly partial multivariate real function: that is, a mapping \( f \) from a subset \( D \) of \( \mathbb{R}^n \) to \( \mathbb{R}^m \) for some integers \( n \geq 0, m > 0 \). The function is called a scalar function if \( m = 1 \), otherwise a vector function. When not otherwise specified, scalar is assumed. The set \( D \) where \( f \) is defined is its domain, also written Domain \( f \). To specify \( n \), call \( f \) an \( n \)-variable point function, or denote values of \( f \) as

\[
f(x_1, \ldots, x_n).
\]

The range of \( f \) over an arbitrary subset \( s \) of \( \mathbb{R}^n \) is the set

\[
\text{Range}(f \mid s) := \{ f(x) \mid x \in s \text{ and } x \in \text{Domain } f \}.
\]

Thus mathematically, when evaluating a function over a set, points outside the domain are ignored (e.g., \( \text{Range}(\sqrt{\mid [-1, 1]} = [0, 1] \)).

Equivalently, for the case where \( f \) takes separate arguments \( s_1, \ldots, s_n \), each being a subset of \( \mathbb{R} \), the range is written as \( \text{Range}(f \mid s_1, \ldots, s_n) \). This is an alternative notation when \( s \) is the cartesian product of the \( s_i \).

A (point) arithmetic operation is a function for which an implementation provides versions in a collection of user-available operations called its library. This includes functions normally written in operator form (e.g., +, \times) and those normally written in function form (e.g., exp, arctan). It is not specified how an implementation provides library facilities.

7.4.3. Interval-valued functions. A box is an interval vector \( x = (x_1, \ldots, x_n) \in \mathbb{IR}^n \). It is usually identified with the cartesian product \( x_1 \times \cdots \times x_n \subseteq \mathbb{R}^n \); however, the correspondence is one-to-one only when all the \( x_j \) are nonempty.

Given an \( n \)-variable scalar point function \( f \), an interval extension of \( f \) is a (total) mapping \( f \) from \( n \)-dimensional boxes to intervals, that is \( f : \mathbb{IR}^n \rightarrow \mathbb{IR} \), such that \( f(x) \in f(x) \) whenever \( x \in x \) and \( f(x) \) is defined, equivalently

\[
f(x) \supseteq \text{Range}(f \mid x).
\]
The object $f$ where

$$f = (e^x + e^{-x})/(2y),$$

is an arithmetic expression in two variables $x, y$ that may be written in the above form as

$$\text{div}(\text{add}(\text{exp}(x), \text{exp}(\text{neg}(x))), \text{mul}(2(), y)),$$

where neg, add, mul, div and exp name the arithmetic operations unary minus, $+$, $\times$, $\div$ and exponential function. The literal constant 2 is regarded as a zero-argument arithmetic operation $2()$, see §7.4.4.

The expression may be split (e.g., by a compiler) into simple assignments each involving a single operation, that may be sequenced in several ways. In the example above, one of these is

$$f = (e^x + e^{-x})/(2y)$$

for any box $x \in \mathbb{R}^n$, regarded as a subset of $\mathbb{R}^n$. The natural interval extension of $f$ is defined by

$$f(x) := \text{hull}(\text{Range}(f | x)).$$

Equivalently, using multiple-argument notation for $f$, an interval extension satisfies

$$f(x_1, \ldots, x_n) \supseteq \text{Range}(f | x_1, \ldots, x_n),$$

and the natural interval extension is defined by

$$f(x_1, \ldots, x_n) := \text{hull}(\text{Range}(f | x_1, \ldots, x_n))$$

for any intervals $x_1, \ldots, x_n$.

In some contexts it is useful for $x$ to be a general subset of $\mathbb{R}^n$, or the $x_i$ to be general subsets of $\mathbb{R}$; the definition is unchanged.

The natural extension is automatically defined for all interval or set arguments. (The decoration system introduced below in §7.8 gives a systematic way of diagnosing when the underlying point function has been evaluated outside its domain.)

When $f$ is a binary operator $\bullet$ written in infix notation, this gives the usual definition of its natural interval extension as

$$x \bullet y = \text{hull}(\{ x \bullet y \mid x \in x, y \in y \}, \text{and } x \bullet y \text{ is defined}).$$

[Example. With these definitions, the relevant natural interval extensions satisfy

$$\sqrt{-1, 4} = [0, 2] \text{ and } \sqrt{-2, -1} = \emptyset; \text{ also } x \times [0, 0] = [0, 0] \text{ for any nonempty } x, \text{ and } x/[0, 0] = \emptyset, \text{ for any } x.]

When $f$ is a vector point function, a vector interval function with the same number of inputs and outputs as $f$ is called an interval extension of $f$ if each of its components is an interval extension of the corresponding component of $f$.

An interval-valued function in the library is called an interval arithmetic operation if it is an interval extension of a point arithmetic operation, and an interval non-arithmetic operation otherwise. Examples of the latter are interval intersection and union, $(x, y) \mapsto x \cap y$ and $(x, y) \mapsto \text{hull}(x \cup y)$.

7.4.4. Constants. A real scalar function with no arguments—a mapping $\mathbb{R}^n \to \mathbb{R}^m$ with $n = 0$ and $m = 1$—is a real constant. Languages may distinguish between a literal constant (e.g., the decimal value defined by the string $1.23e4$) and a named constant (e.g., $\pi$) but the difference is not relevant on Level 1 (and easily handled by outward rounding on Level 2).

From the definition, an interval extension of a real constant is any zero-argument interval function that returns an interval containing $c$. The natural extension returns the interval $[c, c]$.  

7.5. Expressions and the functions they define.

7.5.1. Expressions. A variable is a symbolic name. A scalar expression in zero or more (independent) variables is a symbolic object defined to be either one of those variables or, recursively, of the form $\phi(g_1, \ldots, g_k)$ where $\phi$ is a symbolic arithmetic or non-arithmetic operation that takes $k$ arguments ($k \geq 0$) and returns a single result, and the $g_i$ are expressions. Other syntax may be used, such as traditional algebraic notation and/or program pseudo-code. An arithmetic expression is one in which all the operations are arithmetic operations.

A vector-valued expression is regarded, for purposes of definition in this standard, as a tuple of scalar expressions (see second example below).

[Examples.]

- The object $f$ where

$$f = (e^x + e^{-x})/(2y)$$

is an arithmetic expression in two variables $x, y$ that may be written in the above form as

$$\text{div}(\text{add}(\text{exp}(x), \text{exp}(\text{neg}(x))), \text{mul}(2(), y)),$$

where neg, add, mul, div and exp name the arithmetic operations unary minus, $+$, $\times$, $\div$ and exponential function. The literal constant 2 is regarded as a zero-argument arithmetic operation $2()$, see §7.4.4.

The expression may be split (e.g., by a compiler) into simple assignments each involving a single operation, that may be sequenced in several ways. In the example above, one of these is

$$f = (e^x + e^{-x})/(2y)$$

for any box $x \in \mathbb{R}^n$, regarded as a subset of $\mathbb{R}^n$. The natural interval extension of $f$ is defined by

$$f(x) := \text{hull}(\text{Range}(f | x)).$$

Equivalently, using multiple-argument notation for $f$, an interval extension satisfies

$$f(x_1, \ldots, x_n) \supseteq \text{Range}(f | x_1, \ldots, x_n),$$

and the natural interval extension is defined by

$$f(x_1, \ldots, x_n) := \text{hull}(\text{Range}(f | x_1, \ldots, x_n))$$

for any intervals $x_1, \ldots, x_n$.

In some contexts it is useful for $x$ to be a general subset of $\mathbb{R}^n$, or the $x_i$ to be general subsets of $\mathbb{R}$; the definition is unchanged.

The natural extension is automatically defined for all interval or set arguments. (The decoration system introduced below in §7.8 gives a systematic way of diagnosing when the underlying point function has been evaluated outside its domain.)

When $f$ is a binary operator $\bullet$ written in infix notation, this gives the usual definition of its natural interval extension as

$$x \bullet y = \text{hull}(\{ x \bullet y \mid x \in x, y \in y \}, \text{and } x \bullet y \text{ is defined}).$$

[Example. With these definitions, the relevant natural interval extensions satisfy

$$\sqrt{-1, 4} = [0, 2] \text{ and } \sqrt{-2, -1} = \emptyset; \text{ also } x \times [0, 0] = [0, 0] \text{ for any nonempty } x, \text{ and } x/[0, 0] = \emptyset, \text{ for any } x.]

When $f$ is a vector point function, a vector interval function with the same number of inputs and outputs as $f$ is called an interval extension of $f$ if each of its components is an interval extension of the corresponding component of $f$.

An interval-valued function in the library is called an interval arithmetic operation if it is an interval extension of a point arithmetic operation, and an interval non-arithmetic operation otherwise. Examples of the latter are interval intersection and union, $(x, y) \mapsto x \cap y$ and $(x, y) \mapsto \text{hull}(x \cup y)$.

7.4.4. Constants. A real scalar function with no arguments—a mapping $\mathbb{R}^n \to \mathbb{R}^m$ with $n = 0$ and $m = 1$—is a real constant. Languages may distinguish between a literal constant (e.g., the decimal value defined by the string $1.23e4$) and a named constant (e.g., $\pi$) but the difference is not relevant on Level 1 (and easily handled by outward rounding on Level 2).

From the definition, an interval extension of a real constant is any zero-argument interval function that returns an interval containing $c$. The natural extension returns the interval $[c, c]$.  

7.5. Expressions and the functions they define.

7.5.1. Expressions. A variable is a symbolic name. A scalar expression in zero or more (independent) variables is a symbolic object defined to be either one of those variables or, recursively, of the form $\phi(g_1, \ldots, g_k)$ where $\phi$ is a symbolic arithmetic or non-arithmetic operation that takes $k$ arguments ($k \geq 0$) and returns a single result, and the $g_i$ are expressions. Other syntax may be used, such as traditional algebraic notation and/or program pseudo-code. An arithmetic expression is one in which all the operations are arithmetic operations.

A vector-valued expression is regarded, for purposes of definition in this standard, as a tuple of scalar expressions (see second example below).

[Examples.]

- The object $f$ where

$$f = (e^x + e^{-x})/(2y)$$

is an arithmetic expression in two variables $x, y$ that may be written in the above form as

$$\text{div}(\text{add}(\text{exp}(x), \text{exp}(\text{neg}(x))), \text{mul}(2(), y)),$$

where neg, add, mul, div and exp name the arithmetic operations unary minus, $+$, $\times$, $\div$ and exponential function. The literal constant 2 is regarded as a zero-argument arithmetic operation $2()$, see §7.4.4.

The expression may be split (e.g., by a compiler) into simple assignments each involving a single operation, that may be sequenced in several ways. In the example above, one of these is
All these forms are regarded as defining the same expression \( f \).

- A vector expression that returns the two roots of \( ax^2 + bx + c = 0 \) is regarded for definitional purposes as the pair of expressions \((-b - \sqrt{b^2 - 4ac})/2a\) and \((-b + \sqrt{b^2 - 4ac})/2a\), although in practical code it would be simplified so that common subexpressions are only evaluated once.

- An expression that uses the interval intersection or union operation is a non-arithmetic expression.

To define functions of variables, an expression must be made into a **bound expression** by giving it a *formal argument list* with notation such as

\[
f(z_1, \ldots, z_n) = \text{expression}.
\]

This defines \( f \) to be the indicated expression with the formal argument list \( z_1, \ldots, z_n \), which must include at least the variables that actually occur in the right-hand side. An expression without such an argument list is a **free expression**.

7.5.2. **Generic functions.** An arithmetic operation name such as +, or a bound arithmetic expression such as \( f(x, y) = x + \sin y \), may be used to refer to different, related, functions: such operations and expressions, or the resulting functions, are called generic (or polymorphic). Whether generic function syntax can be used in program code is language-defined.

An implementation provides a number of arithmetic operation names \( \phi \), each representing various functions as follows.

(a) The point function of \( \phi \). This is unique, theoretical and generally non-representable in finite precision.

(b) Various **interval versions** of \( \phi \), among them in particular
   - (b1) The natural interval extension of the point function. This also is unique, theoretical and generally non-representable in finite precision.
   - (b2) Computable interval extensions of the point function.

(c) (Decorated interval versions of \( \phi \), see §7.8)→\text{delete}\[3\] An implementation’s library by definition comprises all its computable versions of required or recommended operations that it provides for any of its supported interval types, as specified in §7.6; §7.7 and in Clause 8.

An arithmetic operation or expression may also denote a floating point function; these are not defined by this standard.

7.5.3. **Point functions and interval versions of expressions.** A bound arithmetic expression \( f = f(z_1, \ldots, z_n) \) represents various functions in the categories (a), (b1), (b2), (c)→\text{delete}\[4\] above.

The **point function** of \( f \) (or defined by) \( f \) is a function \( f : \text{Domain } f \subseteq \mathbb{R}^n \rightarrow \mathbb{R} \). The **natural domain** Domain \( f \) of \( f \) is the set of all \( x = (x_1, \ldots, x_n) \in \mathbb{R}^n \) where its value is defined according to the following rules:

- If \( f \) is the variable \( x_i \), \( f(x) \) is the number \( x_i \). It is defined for all \( x \in \mathbb{R}^n \).

- Recursively, if \( f = \phi(g_1, \ldots, g_k) \), \( f(x) \) is the value of the point function of \( \phi \) at \( u = (u_1, \ldots, u_k) \) where \( u_i \) is the value of the point function of \( g_i \) at \( x \). It is defined for those \( x \in \mathbb{R}^n \) such that each of \( u_1, \ldots, u_k \) is defined and the resulting \( u \) is in \( \phi \)'s domain Domain \( \phi \), which is part of its mathematical definition.

In the recursive clause of this definition, \( \phi \) can be a constant (a function with \( k = 0 \) arguments), so that \( f(x) = \phi() \). Then, see §7.4.4, there are two cases: (a) \( \phi \) has a real value \( c \); then \( f(x) = c \) for all \( x \in \mathbb{R}^n \); (b) \( \phi \) is the undefined function NaN, in which case \( f(x) \) is not defined for any \( x \in \mathbb{R}^n \).

---

3\text{CK 2011-10-15 Not part of the motion}
4\text{CK 2011-10-15 Not part of the motion}
Example. The specifications
\[
\begin{align*}
  f(x, y) &= (e^x + e^{-x})/(2y), \\
  f(y, x) &= (e^x + e^{-x})/(2y), \\
  f(w, x, y, z) &= (e^x + e^{-x})/(2y)
\end{align*}
\]
are all valid and different (they define different functions), while
\[
f(y) = (e^x + e^{-x})/(2y)
\]
is invalid since the argument list does not include the variable \( x \), which occurs in \( f \).

Example. The natural domain of \( f(x, y) = 1/(\sqrt{x - 1} - y) \) is the set of \((x, y)\) in the plane that do not cause either square root of a negative number or division by zero, i.e., where \( x \geq 1 \) and \( y \neq \sqrt{x - 1} \).

An interval version of \( f(z_1, \ldots, z_n) \) is a (total) function \( f : \mathbb{I} \mathbb{R}^n \rightarrow \mathbb{I} \mathbb{R} \). Its value at an actual argument \( x = (x_1, \ldots, x_n) \in \mathbb{I} \mathbb{R}^n \), a box, is recursively constrained as follows:
- If \( f \) is the variable \( z_i \), the value is some interval containing \( x_i \).
- If \( f = \phi(g_1, \ldots, g_k) \), the value is some interval extension of \( \phi \) evaluated at \((u_1, \ldots, u_k)\) where \( u_i \) is the value of some interval version of \( g_i \) at \( x \).

The natural interval version of \( f \) is the unique interval version such that when \( f \) is \( z_i \), the value equals \( x_i \), and that uses the natural interval extension of each \( \phi \).

Moore's theorem states:

**Theorem 7.1 (Fundamental Theorem of Interval Arithmetic, FTIA).**

(i) Every interval version of an arithmetic expression \( f(z_1, \ldots, z_n) \) is an interval extension of the point function defined by the expression.

(ii) If every variable occurs at most once in \( f \) and, for some \( x \in \mathbb{I} \mathbb{R}^n \), all elementary operations occurring in \( f \) evaluate to their range, then \( f(x) = \text{Range}(f | x) \).

[Note. To part (ii). Using exact arithmetic, that is, the natural interval version of \( f \), it is known that an elementary operation evaluates to its range if it is continuous on its input box. (This can be checked ! by the decoration system \( \S 7.8 \), so the conditions of (ii) are computable. The result is that in finite precision, it is often verifiable at run time that \( f(x) \) equals \( \text{Range}(f | x) \) up to roundoff error.)]
7.6. Required operations.

For the interval-valued functions listed in this subclause, an implementation shall provide interval versions appropriate to its supported interval types. For constants and the forward and reverse arithmetic operations in §7.6.1, 7.6.2, 7.6.3, 7.6.4, each interval version shall be an interval extension of the corresponding point function—for a constant, that means any constant interval enclosing the point value. The required rounding behavior of these, and of the numeric functions of intervals in §7.6.8, is detailed in §8.9 8.11.

The names of operations in this standard, as well as symbols used for operations (e.g., for the comparisons in §7.6.9), may not correspond to those that any particular language would use.

7.6.1. Interval literals.

An implementation shall provide denotations of exact interval values by text strings. These are called interval literals. Level 1, which is mainly for human communication, makes no requirements on the form of literals. This document uses the Level 2 syntax, specified in §8.11.1.

[Example. This includes the inf-sup form $[1.234e5, \text{Inf}]$; the mid-rad form $<3.1416+-0.00001>$; or the named interval constant $\text{Entire}$.]

An invalid denotation has no value at Level 1.

7.6.2. Forward-mode elementary functions.

Table on page 19 lists required arithmetic operations. The term operation includes functions normally written in function notation $f(x, y, \ldots)$, as well as those normally written in unary or binary operator notation, $\cdot x$ or $x \cdot y$.

[Note. The list includes all general-computational operations in 754 §5.4 except convertFromInt, and some recommended functions in 754 §9.2.]

Proving the correctness of interval computations relies on the Fundamental Theorem of Interval Arithmetic, which in turn relies on the relation between a point-function and its interval versions. Thus the domain of each point-function and its value at each point of the domain are specified to aid a rigorous implementation. This is mostly straightforward but needs care for functions with discontinuities, such as pow() and atan2().

⚠️ We probably should have a version of interval division $x/y$ that returns two intervals so that it doesn’t lose information when 0 is an interior point of $y$. Several solutions exist, e.g. Kulisch’s; Kearfott’s, which is similar; and Vienna proposal’s “division with gap”. A motion please!
Table 1. Required forward elementary functions.

Normal mathematical notation is used to include or exclude an interval endpoint, e.g., \((-\pi, \pi]\) denotes \(\{x \in \mathbb{R} \mid -\pi < x \leq \pi\}\).

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Point function domain</th>
<th>Point function range</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>neg((x))</td>
<td>(-x)</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{R})</td>
<td></td>
</tr>
<tr>
<td>add((x,y))</td>
<td>(x+y)</td>
<td>(\mathbb{R}^2)</td>
<td>(\mathbb{R})</td>
<td></td>
</tr>
<tr>
<td>sub((x,y))</td>
<td>(x-y)</td>
<td>(\mathbb{R}^2)</td>
<td>(\mathbb{R})</td>
<td></td>
</tr>
<tr>
<td>mul((x,y))</td>
<td>(xy)</td>
<td>(\mathbb{R}^2)</td>
<td>(\mathbb{R})</td>
<td></td>
</tr>
<tr>
<td>div((x,y))</td>
<td>(x/y)</td>
<td>(\mathbb{R}^2 \setminus {y = 0})</td>
<td>(\mathbb{R})</td>
<td>a</td>
</tr>
<tr>
<td>recip((x))</td>
<td>(1/x)</td>
<td>(\mathbb{R} \setminus {0})</td>
<td>(\mathbb{R} \setminus {0})</td>
<td></td>
</tr>
<tr>
<td>sqrt((x))</td>
<td>(\sqrt{x})</td>
<td>([0, \infty))</td>
<td>([0, \infty))</td>
<td>b</td>
</tr>
<tr>
<td>fma((x,y,z))</td>
<td>((x \times y) + z)</td>
<td>(\mathbb{R}^3)</td>
<td>(\mathbb{R})</td>
<td></td>
</tr>
<tr>
<td>case((c,g,h))</td>
<td></td>
<td></td>
<td></td>
<td>See §7.6.3</td>
</tr>
<tr>
<td>sqrt((x))</td>
<td>(x^2)</td>
<td>(\mathbb{R})</td>
<td>((0, \infty))</td>
<td></td>
</tr>
<tr>
<td>pown((x,p))</td>
<td>(x^p), (p \in \mathbb{Z})</td>
<td>({\mathbb{R} \text{ if } p \geq 0} \setminus {0} \text{ if } p &lt; 0)</td>
<td>({0, \infty} \text{ if } p &gt; 0 \text{ odd})</td>
<td>b</td>
</tr>
<tr>
<td>pow((x,y))</td>
<td>(x^y)</td>
<td>({x&gt;0} \cup {x=0, y&gt;0})</td>
<td>((0, \infty))</td>
<td>a</td>
</tr>
<tr>
<td>exp,exp2,exp10((x))</td>
<td>(b^x)</td>
<td>(\mathbb{R})</td>
<td>((0, \infty))</td>
<td>d</td>
</tr>
<tr>
<td>log,log2,log10((x))</td>
<td>(\log_b x)</td>
<td>((0, \infty))</td>
<td>(\mathbb{R})</td>
<td>a</td>
</tr>
<tr>
<td>sin((x))</td>
<td>(\mathbb{R})</td>
<td>([-1, 1])</td>
<td>([0, \infty))</td>
<td>b</td>
</tr>
<tr>
<td>cos((x))</td>
<td>(\mathbb{R})</td>
<td>([-1, 1])</td>
<td>([0, \infty))</td>
<td>b</td>
</tr>
<tr>
<td>tan((x))</td>
<td>(\mathbb{R} \setminus {(k + \frac{1}{2})\pi</td>
<td>k \in \mathbb{Z}})</td>
<td>(\mathbb{R})</td>
<td>([0, \infty))</td>
</tr>
<tr>
<td>asin((x))</td>
<td>([-1, 1])</td>
<td>([-\pi/2, \pi/2])</td>
<td>([0, \infty))</td>
<td>b</td>
</tr>
<tr>
<td>acos((x))</td>
<td>([-1, 1])</td>
<td>([0, \pi])</td>
<td>([0, \infty))</td>
<td>b</td>
</tr>
<tr>
<td>atan((y,x))</td>
<td>(\mathbb{R})</td>
<td>((-\pi/2, \pi/2))</td>
<td>([0, \infty))</td>
<td>b</td>
</tr>
<tr>
<td>atan2((y,x))</td>
<td>(\mathbb{R}^2 \setminus {(0,0)})</td>
<td>((-\pi, \pi])</td>
<td>([0, \infty))</td>
<td>b</td>
</tr>
<tr>
<td>sinh((x))</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{R})</td>
<td>([0, \infty))</td>
<td>b</td>
</tr>
<tr>
<td>cosh((x))</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{R})</td>
<td>([1, \infty))</td>
<td>b</td>
</tr>
<tr>
<td>tanh((x))</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{R})</td>
<td>((-1, 1))</td>
<td>b</td>
</tr>
<tr>
<td>asinh((x))</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{R})</td>
<td>([1, \infty))</td>
<td>b</td>
</tr>
<tr>
<td>acosh((x))</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{R})</td>
<td>([0, \infty))</td>
<td>b</td>
</tr>
<tr>
<td>atanh((x))</td>
<td>((-1, 1))</td>
<td>(\mathbb{R})</td>
<td>([-1, 0, 1))</td>
<td>b</td>
</tr>
<tr>
<td>sign((x))</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{R})</td>
<td>([-1, 0, 1))</td>
<td>b</td>
</tr>
<tr>
<td>ceil((x))</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{Z})</td>
<td>(\mathbb{Z})</td>
<td>b</td>
</tr>
<tr>
<td>floor((x))</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{Z})</td>
<td>(\mathbb{Z})</td>
<td>b</td>
</tr>
<tr>
<td>round((x))</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{Z})</td>
<td>(\mathbb{Z})</td>
<td>b</td>
</tr>
<tr>
<td>roundTiesToEven((x))</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{Z})</td>
<td>(\mathbb{Z})</td>
<td>b</td>
</tr>
<tr>
<td>roundTiesToAway((x))</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{Z})</td>
<td>(\mathbb{Z})</td>
<td>b</td>
</tr>
<tr>
<td>trunc((x))</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{Z})</td>
<td>(\mathbb{Z})</td>
<td>b</td>
</tr>
<tr>
<td>abs((x))</td>
<td>(</td>
<td>x</td>
<td>)</td>
<td>(\mathbb{R})</td>
</tr>
<tr>
<td>min((x_1, . . . , x_k))</td>
<td>(\mathbb{R}^k) for (k = 2, 3, . . .)</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{R})</td>
<td>a</td>
</tr>
<tr>
<td>max((x_1, . . . , x_k))</td>
<td>(\mathbb{R}^k) for (k = 2, 3, . . .)</td>
<td>(\mathbb{R})</td>
<td>(\mathbb{R})</td>
<td>a</td>
</tr>
</tbody>
</table>

Notes to Table 1

a. In describing the domain, notation such as \(\{y = 0\}\) is short for \(\{(x, y) \in \mathbb{R}^2 \mid y = 0\}\), etc.
b. Regarded as a family of functions parameterized by the integer argument \(p\).
c. Defined as \(e^{\text{yint}x}\) for real \(x > 0\) and all real \(y\), and 0 for \(x = 0\) and \(y > 0\), else there is no value at Level 1.
d. \(b = e, 2\) or 10, respectively.
e. The ranges shown are the mathematical range of the point function. To ensure containment, an interval result may include values just outside the mathematical range.
f. \(\text{atan2}(y,x)\) is the principal value of the argument (polar angle) of \((x,y)\) in the plane.
To avoid confusion with notation for open intervals, in this table coordinates in \( \mathbb{R}^2 \) are delimited by angle brackets (\( \langle \rangle \)).

h. \( \text{sign}(x) \) is \(-1\) if \( x < 0 \); \( 0 \) if \( x = 0 \); and \( 1 \) if \( x > 0 \).

i. \( \text{ceil}(x) \) is the smallest integer \( \geq x \). \( \text{floor}(x) \) is the largest integer \( \leq x \). \( \text{roundToEven}(x), \text{roundToAway}(x) \) are the nearest integer to \( x \), with ties rounded to the even integer or away from zero respectively.

\( \text{trunc}(x) \) is the nearest integer to \( x \) in the direction of zero. (As defined in the C standard §7.12.9.)

j. Smallest, or largest, of its real arguments. A family of functions parameterized by the arity \( k \).

7.6.3. Interval case expressions and case function.

Functions are often defined by conditionals: \( f(x) \) equals \( g(x) \) if some condition on \( x \) holds, and \( h(x) \) otherwise. To handle interval extensions of such functions in a way that automatically conforms to the Fundamental Theorem of Interval Arithmetic, the ternary function \( \text{case}(c, g, h) \) is provided. To simplify defining its interval extension, the argument \( c \) specifying the condition is real (instead of boolean), and the condition means \( c < 0 \) by definition. That is,

\[
\text{case}(c, g, h) = \begin{cases} 
  g, & \text{if } c < 0 \\
  h, & \text{else}
\end{cases}
\]

An implementation shall provide the following (see the Notes) interval extension:

\[
\text{case}(c, g, h) = \begin{cases} 
  \emptyset, & \text{if } c \text{ is empty} \\
  g, & \text{if } c < 0 \\
  h, & \text{if } c \geq 0 \\
  \text{convexHull}(g, h), & \text{else}
\end{cases}
\]  

(6)

for any intervals \( c, g, h \), where \( c = [c, \bar{c}] \) when nonempty.

The function \( f \) above may be encoded as \( f(x) = \text{case}(c(x), g(x), h(x)) \). Then, if \( c, g, h \) are interval functions that are interval extensions of point functions \( c, g \) and \( h \), the function

\[
f(x) = \text{case}(c(x), g(x), h(x))
\]

(7)

is automatically an interval extension of \( f \).

[Notes]

1. Equation (4) does not define the natural interval extension, which returns Empty if any of its input arguments is empty. Its advantage is that for a function defined by a conditional expression, such as (7), it allows “short-circuiting”. That is, one can suppress evaluation of \( h(x) \) if \( \bar{c} < 0 \), and of \( g(x) \) if \( c \geq 0 \). This is not so for the natural extension.

2. This method is less awkward than using interval comparisons as a mechanism for handling such functions. However, the resulting interval function is usually not the tightest extension of the corresponding point function. E.g., the (point) absolute value \( |x| \) may be defined by

\[
|x| = \text{case}(x, -x, x).
\]

Then it is easy to see that formula (7), applied to a nonempty \( x = [x, \bar{x}] \), gives the exact range \( \{ |x| \mid x \in \mathbb{R} \} \) when \( \bar{x} < 0 \) or \( 0 \leq x \), but the poor enclosure \( (-x) \cup x \) when \( x < 0 \leq \bar{x} \).

3. \( \text{case}(c, g, h) \) is equivalent to the \( C \) expression \( (c < 0 \implies g) \).

4. Compound conditions may be expressed using the \( \text{max} \) and \( \text{min} \) operations: e.g., a real function \( f(x, y) \) that equals \( \sin(xy) \) in the positive quadrant of the plane, and zero elsewhere, may be written

\[
f(x, y) = \text{case}(\min(x, y), 0, \sin(xy)),
\]

since \( \min(x, y) < 0 \) is equivalent to \( (x < 0 \text{ or } y < 0) \).

7.6.4. Reverse-mode elementary functions.

Constraint-satisfaction algorithms use the functions in this subclause for iteratively tightening an enclosure of a solution to a system of equations.

Given a unary arithmetic operation \( \phi \), a reverse interval extension of \( \phi \) is a binary interval function \( \phiRev \) such that

\[
\phiRev(c, x) \supseteq \{ x \in \mathbb{R} \mid \phi(x) \text{ is defined and in } c \},
\]

(8)

for any intervals \( z, x \).
Evaluating all these sums independently costs \( O(s) \) that one knows that forgets to test. With Kaucher/modal intervals a different choice may be appropriate. MUST test for definedness, and making it always defined leads to un-noticed wrong results from code.

One might think it suffices to apply the operation without the optional argument and intersect the result with \( x \). This is less effective because “hull” and “intersect” do not commute. E.g., in the previous example:

\[
\text{hull}([0, 1, 2]) = \text{hull}([0, 1, 2] \cap [-2, -1] \cup [1, 2]) = [1, 1.2].
\]

so no tightening of the enclosure \( x \) is obtained.

\section{7.6.5. Cancellative addition and subtraction.}

\[\text{Answered.} \]

I have made this only apply to bounded intervals, since it really seems hard to frame a specification for unbounded ones that translates unambiguously to the finite precision (Level 2) situation. Also I have deviated from the Motion 12 spec, by making the operations defined only when \( \text{width}(x) \geq \text{width}(y) \). IMO it is actively harmful in applications to make it always defined. This is for the same reasons that it is harmful to replace \( \sqrt{x} \) by the everywhere defined \( \sqrt{|x|} \): an application MUST test for definedness, and making it always defined leads to un-noticed wrong results from code that forgets to test. With Kaucher/modal intervals a different choice may be appropriate.

Cancellative subtraction solves the problem: Recover interval \( z \) from intervals \( x \) and \( y \), given that one knows \( x \) was obtained as the sum \( y + z \).

\[\text{Example. In some applications one has a list of intervals} \ a_1, \ldots, a_n, \text{and needs to form each interval} \ s_k \text{which is the sum of all the} \ a_i \text{except} \ a_k, \text{that is} \ s_k = \sum_{i=1, i\neq k}^n a_i, \text{for} \ k = 1, \ldots, n. \text{Evaluating all these sums independently costs} \ O(n^2) \text{work. However, if one forms the sum} \ s \text{of all the} \ a_i, \text{one can obtain each} \ s_k \text{from} \ s \text{and} \ a_k \text{by cancellative subtraction. This method only costs} \ O(n) \text{work.} \]

\begin{table}[h]
\centering
\caption{Required reverse elementary functions.}
\begin{tabular}{|c|c|}
\hline
From unary functions & From binary functions \\
\hline
sqrRev(c, x) & mulRev(b, c, x) \\
recipRev(c, x) & divRev1(b, c, x) \\
absRev(c, x) & divRev2(a, c, x) \\
powRev(c, x, p) & powRev1(b, c, x) \\
sinRev(c, x) & powRev2(a, c, x) \\
cosRev(c, x) & atan2Rev1(b, c, x) \\
tanRev(c, x) & atan2Rev2(a, c, x) \\
coshRev(c, x) & \hline
\end{tabular}
\end{table}
This example illustrates that in finite precision, computing \( x \) (as a sum of terms) typically incurs at least one roundoff error, and may incur many. Thus the model underlying these cancellative operations is that \( x \) is an enclosure of an unknown true sum \( x_0 \), whereas \( y \) is “exact”. The computed \( z \) is thus an enclosure of an unknown true \( z_0 \) such that \( y + z_0 = x_0 \).

There is an operation `cancelPlus(x, y)`, equivalent to `cancelMinus(x, -y)` and therefore not specified separately.

There is an operation `cancelMinus(x, y)` that returns for any two bounded intervals \( x \) and \( y \) the tightest interval \( z \) such that

\[
y + z \supseteq x
\]

if such a \( z \) exists. Otherwise `cancelMinus(x, y)` has no value at Level 1.

This specification leads to the following Level 1 algorithm. If \( x = \emptyset \) then \( z = \emptyset \). If \( x \neq \emptyset \) and \( y = \emptyset \) then \( z \) has no value. If \( x = [x_l, x_u] \) and \( y = [y_l, y_u] \) are both nonempty and bounded, define \( \bar{z} = x - y \) and \( \bar{z} = \bar{x} - \bar{y} \). Then \( z \) is defined to be \( [\bar{z}, \bar{z}] \) if \( \bar{z} \leq \bar{z} \) (equivalently if \( \text{width}(x) \geq \text{width}(y) \)), and has no value otherwise. If either \( x \) or \( y \) is unbounded, \( z \) has no value.

[Note. Because of the cancellative nature of these operations, care is needed in finite precision to determine whether the result is defined or not. More details are given at Level 3 in §9.5.]

### 7.6.6. Non-arithmetic operations

The following operations shall be provided, the arguments and result being intervals.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>intersection(x, y)</code></td>
<td>( x \cap y )</td>
</tr>
<tr>
<td><code>convexHull(x, y)</code></td>
<td>interval hull of ( x \cup y )</td>
</tr>
</tbody>
</table>

### 7.6.7. Constructors

An interval constructor by definition is an operation that creates a bare or decorated interval from non-interval data. The following bare interval constructors shall be provided.

The operation `nums2interval(l, u)`, where \( l \) and \( u \) are extended-real values, returns the set \( \{ x \in \mathbb{R} \mid l \leq x \leq u \} \). If (see §7.2) the conditions \( l \leq u \), \( l < +\infty \) and \( u > -\infty \) hold, this set is the nonempty interval \([l, u]\) and the operation is said to `succeed`. Otherwise the operation is said to `fail`, and returns no value.

The operation `text2interval(t)` succeeds and returns the interval denoted by the text string \( t \), if \( t \) denotes an interval. Otherwise, it fails and returns no value.

[Note. Since Level 1 is mainly for human-human communication, any understandable \( t \) is acceptable, e.g. “\([3.1, 4.2]\)” or “\([2\pi, \infty]\)”. Rules for the strings \( t \) accepted at an implementation level are given in the Level 2 Subclause §8.17 on I/O and may optionally be followed.]

The constructor `bareempty()` returns Empty.

The constructor `bareentire()` returns Entire.

### 7.6.8. Numeric functions of intervals

The operations in Table [3] shall be provided, the argument being an interval and the result a number, which for some of the operations may be infinite.

[Note. Implementations are recommended to provide an operation that returns \( \text{mid}(x) \) and \( \text{rad}(x) \) simultaneously.]

### 7.6.9. Boolean functions of intervals

The following operations shall be provided, which return a boolean \( (1 = \text{true}, 0 = \text{false}) \) result.

There is a function `isEmpty(x)`, which returns 1 if \( x \) is the empty set, 0 otherwise. There is a function `isEntire(x)`, which returns 1 if \( x \) is the whole line, 0 otherwise.

There are eight comparison relations, which take two interval inputs and return a boolean result. These are defined in Table [4] in which column three gives the set-theoretic definition, and column four gives an equivalent specification when both intervals are nonempty.

The following table shows what the definitions imply when at least one interval is empty.
TABLE 3. Required numeric functions of an interval $x = [a, b]$. Note $\inf$ can return $-\infty$; each of $\sup$, $\wid$, $\rad$ and $\mag$ can return $+\infty$.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\inf(x)$</td>
<td>lower bound of $x$ if $x$ is nonempty $\inf$ if $x$ is empty</td>
</tr>
<tr>
<td>$\sup(x)$</td>
<td>upper bound of $x$ if $x$ is nonempty $-\infty$ if $x$ is empty</td>
</tr>
<tr>
<td>$\mid(x)$</td>
<td>midpoint $(a + b)/2$ if $x$ is nonempty bounded</td>
</tr>
<tr>
<td>$\wid(x)$</td>
<td>width $b - a$ if $x$ is nonempty</td>
</tr>
<tr>
<td>$\rad(x)$</td>
<td>radius $(b - a)/2$ if $x$ is nonempty</td>
</tr>
<tr>
<td>$\mag(x)$</td>
<td>magnitude $\sup{</td>
</tr>
<tr>
<td>$\mig(x)$</td>
<td>magnitude $\inf{</td>
</tr>
</tbody>
</table>

$\Delta$ I have gone for simplicity. Also I have followed my own view that at Level 1 it is more consistent with math conventions for a function to simply have “no value” outside its domain, than for it to return a special value. At Level 2 this translates into returning a value such as NaN.

TABLE 4. Comparisons for intervals $a$ and $b$. Notation $\forall a$ means “for all $a$ in $a$”, and so on. In column 4, $a = [\alpha, \beta]$ and $b = [\beta, \gamma]$, where $\alpha, \beta$ may be $-\infty$ and $\beta, \gamma$ may be $+\infty$; and $\prec$ is the same as $<$ except that $-\infty < -\infty$ and $+\infty < +\infty$ are true.

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Definition</th>
<th>For $a, b \neq \emptyset$</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEqual($a, b$)</td>
<td>$a = b$</td>
<td>$\forall a \exists_0 a = b \land \forall a \exists_0 b = a$</td>
<td>$a = b \land \beta = b$</td>
<td>$a$ equals $b$</td>
</tr>
<tr>
<td>containedIn($a, b$)</td>
<td>$a \subseteq b$</td>
<td>$\forall a \exists_0 a \subseteq b$</td>
<td>$b \leq a \land \beta \leq \gamma$</td>
<td>$a$ is a subset of $b$</td>
</tr>
<tr>
<td>less($a, b$)</td>
<td>$a &lt; b$</td>
<td>$\forall a \exists_0 a \leq b \land \forall a \exists_0 a \leq b$</td>
<td>$a \leq b \land \beta \leq \gamma$</td>
<td>$a$ is weakly less than $b$</td>
</tr>
<tr>
<td>precedes($a, b$)</td>
<td>$a \prec b$</td>
<td>$\forall a \exists_0 a \leq b \land \forall a \exists_0 a \neq b$</td>
<td>$\pi \leq \beta$</td>
<td>$a$ is to left of but may touch $b$</td>
</tr>
<tr>
<td>isInterior($a, b$)</td>
<td>$a @ b$</td>
<td>$\forall a \exists_0 a \prec b \land \forall a \exists_0 b &lt; a$</td>
<td>$b &lt; \beta \land \pi \prec \gamma$</td>
<td>$a$ is interior to $b$</td>
</tr>
<tr>
<td>strictlyLess($a, b$)</td>
<td>$a &lt; b$</td>
<td>$\forall a \exists_0 a &lt; b \land \forall a \exists_0 a &lt; b$</td>
<td>$b &lt; \beta \land \pi &lt; \gamma$</td>
<td>$a$ is strictly less than $b$</td>
</tr>
<tr>
<td>strictlyPrecedes($a, b$)</td>
<td>$a \prec b$</td>
<td>$\forall a \exists_0 a &lt; b \land \forall a \exists_0 a &lt; b$</td>
<td>$\pi &lt; b \land \beta &lt; \gamma$</td>
<td>$a$ is strictly to left of $b$</td>
</tr>
<tr>
<td>areDisjoint($a, b$)</td>
<td>$a \not\prec b$</td>
<td>$\forall a \exists_0 a \neq b$</td>
<td>$a$ and $b$ are disjoint</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$a \neq \emptyset$</th>
<th>$b \neq \emptyset$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = b$</td>
<td>$0$</td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td>$a \subseteq b$</td>
<td>$1$</td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td>$a \subseteq b$</td>
<td>$0$</td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td>$a \prec b$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>$a @ b$</td>
<td>$1$</td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td>$a &lt; b$</td>
<td>$0$</td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td>$a &lt; b$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>$a \not\prec b$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

[Note.  
- All these relations, except $a \not\prec b$, are transitive for nonempty intervals.  
- The first three are reflexive.  
- $\forall a \exists_0$ uses the topological definition: $b$ is a neighbourhood of each point of $a$. This implies, for instance, that $\forall a \exists_0 (\forall a \exists_0$ $\forall a \exists_0$ is true.  
- In fact all occurrences of $<$ in column 4 of Table 4 can be replaced by $\prec$.]
7.6.10. Dot product function.

For point vectors \( x = (x_1, \ldots, x_n) \) and \( y = (y_1, \ldots, y_n) \) the function \( \text{dotProduct}(x, y) = \sum_{i=1}^{n} x_i y_i \) is defined, but has no requirements at Level 1. It is specified in Subclause 8.11.10 which deals with the Complete Arithmetic datatype and operations.
7.7. Recommended operations (informative).

Language standards should define interval versions of some or all functions in this subclause, and some or all supported types, as is most appropriate to the language.

7.7.1. Forward-mode elementary functions.

The list of recommended functions is in Table 5. Each interval version provided is required to be an interval extension of the point function.

Table 5. Recommended elementary functions.

Normal mathematical notation is used to include or exclude an interval endpoint, e.g., \((-1, 1]\) denotes \(\{x \in \mathbb{R} \mid -1 < x \leq 1\}\).

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Point function domain</th>
<th>Point function range</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{rootn}(x,q)</td>
<td>(q \in \mathbb{Z} \setminus {0}) if (q = 0) odd (\mathbb{R}) if (q &gt; 0) even (\mathbb{R}\setminus {0}) if (q &lt; 0) odd (\mathbb{R}) if (q &lt; 0) even</td>
<td>same as domain</td>
<td>(a)</td>
<td></td>
</tr>
<tr>
<td>\texttt{exp1}(x)</td>
<td>(b^x - 1)</td>
<td>(\mathbb{R})</td>
<td>((-1, \infty))</td>
<td>(b)</td>
</tr>
<tr>
<td>\texttt{logp1}(x)</td>
<td>(\log_b(x + 1))</td>
<td>((-1, \infty))</td>
<td>(\mathbb{R})</td>
<td>(b)</td>
</tr>
<tr>
<td>\texttt{compound1}(x,y)</td>
<td>((1 + x)^y - 1)</td>
<td>({x &gt; -1} \cup {x = -1, y &gt; 0}) ([0, \infty))</td>
<td>(0, \infty))</td>
<td>(c)</td>
</tr>
<tr>
<td>\texttt{hypot}(x,y)</td>
<td>(\sqrt{x^2 + y^2})</td>
<td>(\mathbb{R}^2)</td>
<td>([0, \infty))</td>
<td>(d)</td>
</tr>
<tr>
<td>\texttt{rSqrt}(x)</td>
<td>(1/\sqrt{x})</td>
<td>((0, \infty))</td>
<td>((0, \infty))</td>
<td>(d)</td>
</tr>
<tr>
<td>\texttt{sinPi}(x)</td>
<td>(\sin(\pi x))</td>
<td>(\mathbb{R})</td>
<td>([-1, 1])</td>
<td>(e)</td>
</tr>
<tr>
<td>\texttt{cosPi}(x)</td>
<td>(\cos(\pi x))</td>
<td>(\mathbb{R})</td>
<td>([-1, 1])</td>
<td>(e)</td>
</tr>
<tr>
<td>\texttt{tanPi}(x)</td>
<td>(\tan(\pi x))</td>
<td>(\mathbb{R}\setminus {k + \frac{1}{2} \mid k \in \mathbb{Z}}) (\mathbb{R})</td>
<td>([-1/2, 1/2])</td>
<td>(f)</td>
</tr>
<tr>
<td>\texttt{asinPi}(x)</td>
<td>(\arcsin(x)/\pi)</td>
<td>([-1, 1])</td>
<td>([-1/2, 1/2])</td>
<td>(f)</td>
</tr>
<tr>
<td>\texttt{acosPi}(x)</td>
<td>(\arccos(x)/\pi)</td>
<td>([-1, 1])</td>
<td>([0, 1])</td>
<td>(f)</td>
</tr>
<tr>
<td>\texttt{atanPi}(x)</td>
<td>(\arctan(x)/\pi)</td>
<td>(\mathbb{R})</td>
<td>((-1/2, 1/2))</td>
<td>(f)</td>
</tr>
<tr>
<td>\texttt{atan2Pi}(y,x)</td>
<td>(\arctan2(y,x)/\pi)</td>
<td>(\mathbb{R}^2 \setminus {(0, 0)}) ((-1, 1])</td>
<td>((-1, 1])</td>
<td>(f)</td>
</tr>
</tbody>
</table>

Notes to Table 5

a. Regarded as a family of functions parameterized by the integer arguments \(q\), or \(r\) and \(s\).

b. \(b = c, 2\) or \(10\), respectively.

c. Mathematically unnecessary, but included to let implementations give better numerical behavior for small values of the arguments.

d. In describing domains, notation such as \(\{y = 0\}\) is short for \(\{(x, y) \in \mathbb{R}^2 \mid y = 0\}\), and so on.

e. These functions avoid a loss of accuracy due to \(\pi\) being irrational, cf. Table 1 note e.

f. To avoid confusion with notation for open intervals, in this table coordinates in \(\mathbb{R}^2\) are delimited by angle brackets \(\langle \rangle\).

7.7.2. Extended interval overlapping operation \texttt{overlap}(a,b), also written \texttt{a} \texttt{\oplus} \texttt{b}, arises from the work of J.F. Allen [?] on temporal logic. It may be used as an infrastructure for other interval comparisons. If implemented, it should also be available at user level; how this is done is implementation-defined or language-defined.

Allen identified 13 states of a pair \((a,b)\) of nonempty intervals, which are ways in which they can be related with respect to the usual order \(a < b\) of the reals. Together with three states for when either interval is empty, these define the 16 possible values of \texttt{overlap}(a,b).

To describe the states for nonempty intervals of positive width, it is useful to think of \(b = [b, b]\) (with \(b < b\)) as fixed, while \(a = [a, a]\) (with \(a < a\)) starts far to its left and moves to the right. Its endpoints move continuously with strictly positive velocity. Then, depending on the relative sizes of \(a\) and \(b\), the value of \texttt{a} \texttt{\oplus} \texttt{b} follows a path from left to right through the graph below, whose nodes represent Allen’s 13 states.
For instance “a overlaps b”—equivalently \( a \odot b \) has the value overlaps—is the case \( a < b < \pi < b \).

The three extra values are: bothEmpty when \( a = b = \emptyset \), else firstEmpty when \( a = \emptyset \), secondEmpty when \( b = \emptyset \).

Table 6 shows the 16 states, with the 13 “nonempty” states specified (a) in terms of set membership using quantifiers and (b) in terms of the endpoints \( a, a, b, b \), and also (c) shown diagrammatically.

The set and endpoint specifications remove some ambiguities of the diagram view when one interval shrinks to a single point that coincides with an endpoint of the other. Such a case is allocated to equal when all four endpoints coincide; else to starts, finishes, finishedBy or startedBy as appropriate; never to meets or metBy.

[Note. The 16 state values can be encoded in four bits. However, if they are then translated into patterns \( P \) in a 16-bit word, having one position equal to 1 and the rest zero, one can easily implement interval comparisons by using bit-masks.

For instance, suppose we make the states \( s \) in Table 6’s order correspond to the 16 bits in the word, left-to-right, so \( s = \text{bothEmpty maps to } P(s) = 1000000000000000 \), \( s = \text{firstEmpty maps to } P(s) = 0100000000000000 \) and so on. Consider the relation areDisjoint\((a, b)\). This is true if and only if one or both of \( a \) or \( b \) is empty, or \( a \) is “before” \( b \), or \( a \) is “after” \( b \). That is, iff the logical “and” of \( P(s) \) with the mask disjointMask = 1111000000000001 is not identically zero.

This scheme can be efficiently implemented in hardware, see for instance M. Nehmeier, S. Siegel and J. Wolff von Gudenberg. All the required comparisons in this standard can be implemented in this way, as can be, e.g., the “possibly” and “certainly” comparisons of Sun’s interval Fortran. Thus the overlap operation is a primitive from which it is simple to derive all interval comparisons commonly found in the literature.]

7.7.3. Slope functions.

The functions in Table 7 are the commonest ones needed to efficiently implement improved range enclosures via first- and second-order slope algorithms. They are analytic at \( x = 0 \) after filling in the removable singularity there, where each has the value 1.
Table 6. The 16 states of interval overlapping situations for intervals $a, b$.

Notation $\forall_a$ means “for all $a$ in $a$”, and so on. Phrases within a cell are joined by “and”, e.g. starts is specified by $(a = b \land \pi < \beta)$.

<table>
<thead>
<tr>
<th>State $a \odot b$ is</th>
<th>Set specification</th>
<th>Endpoint specification</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>States with either interval empty</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bothEmpty $a = \emptyset \land b = \emptyset$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>firstEmpty $a = \emptyset \land b \neq \emptyset$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>secondEmpty $a \neq \emptyset \land b = \emptyset$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>States with both intervals nonempty</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>before $\forall_a \forall_b a &lt; b$</td>
<td>$\pi &lt; b$</td>
<td>$\pi &lt; b$</td>
<td></td>
</tr>
<tr>
<td>meets $\forall_a \forall_b a &lt; b$</td>
<td>$\pi &lt; b$</td>
<td>$\pi &lt; b$</td>
<td></td>
</tr>
<tr>
<td>overlaps $\exists_a \forall_b a &lt; b$</td>
<td>$\pi &lt; b$</td>
<td>$\pi &lt; b$</td>
<td></td>
</tr>
<tr>
<td>starts $\exists_a \forall_b a &lt; b$</td>
<td>$\pi &lt; b$</td>
<td>$\pi &lt; b$</td>
<td></td>
</tr>
<tr>
<td>containedBy $\exists_a \forall_b a &lt; b$</td>
<td>$\pi &lt; b$</td>
<td>$\pi &lt; b$</td>
<td></td>
</tr>
<tr>
<td>finishes $\exists_a \forall_b a &lt; b$</td>
<td>$\pi &lt; b$</td>
<td>$\pi &lt; b$</td>
<td></td>
</tr>
<tr>
<td>equal $\forall_a \exists_b a = \beta$</td>
<td>$\pi = \beta$</td>
<td>$\pi = \beta$</td>
<td></td>
</tr>
<tr>
<td>finishedBy $\exists_a \forall_b a &lt; b$</td>
<td>$\pi = \beta$</td>
<td>$\pi = \beta$</td>
<td></td>
</tr>
<tr>
<td>contains $\exists_a \forall_b a &lt; b$</td>
<td>$\pi &lt; b$</td>
<td>$\pi &lt; b$</td>
<td></td>
</tr>
<tr>
<td>startedBy $\exists_a \forall_b a &lt; b$</td>
<td>$\pi &lt; b$</td>
<td>$\pi &lt; b$</td>
<td></td>
</tr>
<tr>
<td>overlappedBy $\exists_a \forall_b a &lt; b$</td>
<td>$\pi &lt; b$</td>
<td>$\pi &lt; b$</td>
<td></td>
</tr>
<tr>
<td>metBy $\forall_b \exists_a a = \beta$</td>
<td>$\pi = \beta$</td>
<td>$\pi = \beta$</td>
<td></td>
</tr>
<tr>
<td>after $\forall_b \exists_a a = \beta$</td>
<td>$\pi &lt; b$</td>
<td>$\pi &lt; b$</td>
<td></td>
</tr>
</tbody>
</table>
Table 7. Recommended slope functions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Point function domain</th>
<th>Point function range</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>expSlope1(x)</td>
<td>$\frac{1}{x}(e^x - 1)$</td>
<td>$\mathbb{R}$</td>
<td>(0, $\infty$)</td>
<td></td>
</tr>
<tr>
<td>expSlope2(x)</td>
<td>$\frac{2}{x^2}(e^x - 1 - x)$</td>
<td>$\mathbb{R}$</td>
<td>(0, $\infty$)</td>
<td></td>
</tr>
<tr>
<td>logSlope1(x)</td>
<td>$\frac{2}{x^2} (\log(1+x) - x)$</td>
<td>$\mathbb{R}$</td>
<td>(0, $\infty$)</td>
<td></td>
</tr>
<tr>
<td>logSlope2(x)</td>
<td>$\frac{3}{x^3} (\log(1+x) - x + \frac{x^2}{2})$</td>
<td>$\mathbb{R}$</td>
<td>(0, $\infty$)</td>
<td></td>
</tr>
<tr>
<td>cosSlope2(x)</td>
<td>$-\frac{2}{x^2} (\cos x - 1)$</td>
<td>$\mathbb{R}$</td>
<td>[0, 1]</td>
<td></td>
</tr>
<tr>
<td>sinSlope3(x)</td>
<td>$\frac{6}{x^3} (\sin x - x)$</td>
<td>$\mathbb{R}$</td>
<td>(0, 1)</td>
<td></td>
</tr>
<tr>
<td>asinSlope3(x)</td>
<td>$\frac{6}{x^3} (\arcsin x - x)$</td>
<td>$[-1, 1]$</td>
<td>[1, $3\pi - 6$]</td>
<td></td>
</tr>
<tr>
<td>atanSlope3(x)</td>
<td>$\frac{3}{x^3} (\arctan x - x)$</td>
<td>$\mathbb{R}$</td>
<td>(0, 1)</td>
<td></td>
</tr>
<tr>
<td>coshSlope2(x)</td>
<td>$\frac{2}{x^2} (\cosh x - 1)$</td>
<td>$\mathbb{R}$</td>
<td>[1, $\infty$]</td>
<td></td>
</tr>
<tr>
<td>sinhSlope3(x)</td>
<td>$\frac{3}{x^3} (\sinh x - x)$</td>
<td>$\mathbb{R}$</td>
<td>[1, $\infty$]</td>
<td></td>
</tr>
</tbody>
</table>
8. Level 2 description

8.1. Level 2 introduction. Objects and operations at Level 2 are said to have finite precision. They are the entities from which implementable interval algorithms may be constructed. Level 2 objects are called datums since the standard deals with numeric functions of intervals (such as the midpoint) and interval functions of numbers (such as the construction of an interval from its lower and upper bounds), this clause involves both numeric and interval datums, as well as the set D of decorations.

Following 754 terminology, numeric (floating point) datums are organized into formats. Interval datums are organized into types. Each format or type is a finite set of datums, with associated operations. The standard defines three kinds of interval type:

- **Bare interval types**, see §8.6, represent finite sets of (mathematical, Level 1) intervals.
- **Decorated interval types**, see §8.7, represent finite sets of decorated intervals.
- **Compressed interval types**, see §8.8, implement compressed decorated interval arithmetic.

An implementation shall support at least one bare interval type. If 754-conforming, it shall support the inf-sup type, see §8.6.2, of at least one of the five basic formats binary32, binary64, binary128, decimal64, and decimal128.

There shall be a one-to-one correspondence between bare interval types and decorated interval types, wherein each bare interval type has a corresponding derived decorated interval type. Beyond this, which types are supported is language- or implementation-defined.

This standard uses the term T-version of an operation, where T is a bare or decorated interval type, to mean a finite-precision approximation to the corresponding Level 1 operation, in which any input or output intervals become T-intervals. This includes the following:

(a) A T-interval extension (§8.9) of one of the required or recommended arithmetic operations of §7.6.

(b) A set operation, such as intersection and convex hull of T-intervals, returning a T-interval.

(c) A function such as the midpoint, whose input is a T-interval and output is a numeric value.

(d) A constructor, whose input is numeric or text and output is a T-interval.

(e) The T-interval hull, regarded as a conversion operation, see §8.8.2.

Generically these comprise the operations of the type T, for the implementation.

An implementation may also support mixed-type operations, where input and output intervals are not all of the same type.

It is language-defined whether the type of a datum can be determined at run time.

8.2. Naming conventions for operations. An operation can exist in many forms at Level 2 depending on the input and result types, and is generally given a name that suits the context.

For example, the addition of two interval datums x, y may be written in generic algebra notation x + y; or with a generic text name addition(x, y); or giving full type information such as decimal64-infsup-addition(x, y).

It may also be written as T-addition(x, y) to show it is an operation of a particular but unspecified type T, or—in the context of 754-conforming types—as typeOf-addition(x, y) where typeOf has a similar meaning to 754's formatOf.

In a specific language or programming environment, the names used for types may differ from those used in this document.

8.3. 754-conformance. This should be enlarged to “Conformance and 754-conformance”?

The standard defines the notion of 754-conformance, whose stronger requirements improve accuracy and programming convenience. In this context a part of an implementation means a subset of the set of supported Level 2 interval types.

A 754-conforming type is an inf-sup type derived from a 754 floating point format—one of the five basic types or an extended precision or extendable precision format—that meets the general requirements for conformance and whose operations meet the accuracy requirements in Table 9.

A 754-conforming part of an implementation is a subset of the set of supported 754-conforming types, that meets the requirements for mixed-type arithmetic in the next paragraphs.

---

Not “data”, whose common meaning could cause confusion.
A **754-conforming implementation** is one where *all* supported types are 754-conforming, and the whole set of supported types meets the requirements for mixed-type arithmetic.

8.3.1. **754-conforming mixed-type arithmetic.** The 754 standard requires a conforming floating point system to provide mixed-format "formatOf" operations. That is, the output format is specified and the inputs may be of any format of the same radix as the output. The result is computed as if using the exact inputs and rounded to the required accuracy on output. This eliminates the problem of double rounding in mixed-format work, which otherwise can cause significant growth of errors.

A 754-conforming part of an implementation shall provide (e.g., by exploiting the formatOf feature at Level 3) corresponding mixed-type “typeOf” operations. That is, the output type is specified and the inputs may be of any type of the same radix as the output. The result shall be computed as if using the exact inputs and shall meet the accuracy requirements for each operation, specified in Table 8.

These requirements shall apply to all Level 2 operations with interval operands, without explicit mention.

8.4. **Datums are tagged by names.** A Level 2 format or type is an abstraction of a particular way to represent numbers or intervals—e.g., “IEEE 64 bit binary” for numbers—focusing on the Level 1 objects represented, and hiding the Level 3 method by which it is done.

However a datum is more than just the Level 1 value: for instance the number 3.75 represented in 64 bit binary is a different datum from the same number represented in 64 bit decimal.

This is achieved by formally regarding each datum as a pair:

- number datum = (Level 1 number, format name),
- interval datum = (Level 1 interval, type name),

where the name is some symbol that uniquely identifies the format or type. The Level 1 value is said to be **tagged** by the name. This achieves two needed properties: distinct formats or types are disjoint sets; and two datums are equal if and only if they represent the same Level 1 value tagged by the same name.

By convention, such names are omitted from datums except when clarity requires. *[Example. Level 2 interval addition within a type named \( t \) is normally written \( z = x + y \), though the full correct form is \( (z, t) = (x, t) + (y, t) \). The full form might be used, for instance, to indicate that mixed-type addition is forbidden between types \( s \) and \( t \) but allowed between types \( s \) and \( u \). Namely, one can say that \( (x, s) + (y, t) \) is undefined, but \( (x, s) + (y, u) \) is defined.]*

A format or type may be given an alternative name, e.g. binary64 might be abbreviated to b64 for convenience.

8.5. **Number formats.** Having regard to 8.4 a **number format**, or just format, is the set of all pairs \( (x, f) \) where \( x \) belongs to a set \( F \), and \( f \) is a name for the format.

\( F \) comprises a finite subset of the extended reals \( \mathbb{R} \), together with a value NaN. A **numeric** member of \( F \) is one that is not NaN. \( F \) shall contain zero, \( -\infty \) and \( +\infty \), and shall be symmetric: if a numeric \( x \) is in \( F \), so is \(-x\).

Following the convention of omitting names, the format is normally identified with the set \( F \), and one may say a number format is a set of datums comprising NaN together with a finite, symmetric, set \( F \) of extended reals that contains 0 and \( \pm\infty \). The non-NaN members of \( F \) are called \( F \)-numbers.

*[Note. At Level 2 each format has only one NaN datum, but this may correspond to more than one NaN at Level 3, e.g. by using the payload of a 754 NaN.]*

A floating-point format in the 754 sense, such as binary64, is identified with the number format for which \( F \) is the set of extended-real numbers that are exactly representable in that format, where \(-0 \) and \(+0 \) both represent the mathematical number 0.

A number format \( F \) is said to be **compatible** with an interval type \( T \), if each non-empty \( T \)-interval contains at least one finite \( F \)-number.

8.6. **Bare interval types.**

8.6.1. **Definition.** Having regard to 8.4 a **bare interval type** is the set of all pairs \( (x, t) \) where \( x \) belongs to a finite subset \( T \) of the mathematical intervals \( \mathbb{IR} \) that contains Empty and Entire, and \( t \) is a name for the type.
Following the convention of omitting names, the type is normally identified with the set \( T \),
and one may say a bare interval type is an arbitrary finite set \( T \) of intervals that contains Empty
and Entire.

A \( T \)-interval means an interval belonging to \( T \); a \( T \)-box means a box with \( T \)-interval component.

Examples. To illustrate the flexibility allowed in defining types, let \( S_1 \) and \( S_2 \) be the sets of inf-sup
intervals using 754 single (binary32) and double (binary64) precision respectively. That is, a member
of \( S_1 \) [respectively \( S_2 \)] is either empty, or an interval whose bounds are exactly representable in binary32
[respectively binary64].

An implementation can (and usually would) define these as different types, by tagging members of
\( S_1 \) by one type name \( t_1 \) and members of \( S_2 \) by another name \( t_2 \). At Level 3 they would be represented
as a pair of binary32 or binary64 floating point datums respectively. However, it could treat them
as one type, with the representation by a pair of binary32’s being a space-saving alternative to the
pair of binary64’s, to be used when convenient.

8.6.2. Inf-sup and mid-rad types. The inf-sup type derived from a given number format \( F \)
(the type \( F \) inf-sup, e.g., “binary64 inf-sup”) is the bare interval type \( T \) comprising all intervals
whose endpoints are in \( F \), together with Empty. \( F \) is termed the parent format of \( T \). Note that
Entire is in \( T \) because \( \pm\infty \in F \) by the definition of a number format, so \( T \) satisfies the requirements
for a bare interval type given in §8.6.1

Mid-rad types are not specified by this standard but are useful for examples. A mid-rad
bare interval type is taken to be one whose nonempty bounded intervals comprise all intervals of the
form \([m-r, m+r] \), where \( m \) is in some number format \( F \), and \( r \) is in a possibly different
number format \( F' \), with \( m, r \) finite and \( r \geq 0 \). From the definition in §8.6.1 the type must also
contain Empty and Entire, so at Level 3 there must be a way to represent these; it may also contain
semi-infinite intervals.

8.7. Multi-precision interval types. Multi-precision floating point systems—extendable
precision in 754 terminology—generally provide an (at least conceptually) infinite sequence of levels
of precision, where there is a finite set \( F_n \) of numbers representable at the \( n \)th level \((n = 1, 2, 3, \ldots)\),
and \( F_1 \subset F_2 \subset F_3 \ldots \). These are typically used to define a corresponding infinite sequence of interval
types \( T_n \) with \( T_1 \subset T_2 \subset T_3 \ldots \).

Example. For multi-precision systems that define a nonempty \( T_n \)-interval to be one whose endpoints
are \( F_n \)-numbers, each \( T_n \) is an inf-sup type with a unique interval hull operation—explicit, in the sense of
§8.8.

A conforming implementation must define such \( T_n \) as a parameterized sequence of interval
types. It cannot take the union over \( n \) of the sets \( T_n \) as a single type, because this infinite set has
no interval hull operation: there is generally no tightest member of it enclosing a given set of real
numbers. This constrains the design of conforming multi-precision interval systems.

8.8. Explicit and implicit types, and Level 2 hull operation.

8.8.1. Hull in one dimension. Each bare interval type \( T \) shall have an interval hull
operation specified:

\[
y = \text{hull}(s),
\]

which is part of its definition and maps an arbitrary set of reals, \( s \), to a minimal \( T \)-interval \( y \)
enclosing \( s \). Minimal is in the sense that

\[
s \subseteq y \quad \text{and for any other } T\text{-interval } z, \text{ if } s \subseteq z \subseteq y \text{ then } z = y.
\]

For clarity when needed, this operation is called the \( T \)-hull and denoted \text{hull}_T.

Since \( T \) is a finite set and contains Entire, such a minimal \( y \) exists for any \( s \). In general \( y \) may
not be unique. If it is unique for every subset \( s \subset \mathbb{R} \), then the type \( T \) is called explicit, otherwise
it is implicit. For an explicit type, the hull operation is uniquely determined and need not be
separately specified. For an implicit type, the implementation’s documentation shall specify the hull
operation, e.g., by an algorithm.

Two types with different hull operations are different, even if they have the same set of intervals.

Examples. It is easy to see that every inf-sup type is explicit. A mid-rad type is typically implicit.

As an example of the need for a specified hull algorithm, let \( T \) be the mid-rad type \((8.6.2)\) where
\( m \) and \( r \) use the same floating point format \( F \), say binary64, and let \( s \) be the interval \([ -1, 1 + \epsilon ] \) where
1 + \epsilon is the next \mathbb{F}-number above 1. Clearly any minimal interval \((m, r)\) enclosing \(s\) has \(r = 1 + \epsilon\). But \(m\) can be any of the many \mathbb{F}-numbers in the range 0 to \(\epsilon\); each of these gives a minimal enclosure of \(s\).

A possible general algorithm, for a bounded set \(s\) and a mid-rad type, is to choose \(m \in \mathbb{F}\) as close as possible to the mathematical midpoint of the interval \([\text{\inf } s, \text{\sup } s]\) (with some way to resolve ties) and then the smallest \(r \in \mathbb{F}\) such that \(r \geq \max(m - \delta, r - m)\). The cost of performing this depends on how the set \(s\) is represented. If \(s\) is a binary64 inf-sup interval, it is simple. If \(s\) is defined as the range of some exotic function, it could be expensive.

For 754-conforming implementations the hull operations of the inf-sup types derived from the formats binary32, binary64, binary128, decimal64 and decimal128 are denoted respectively as

\[
\text{hull}_{b32}, \text{hull}_{b64}, \text{hull}_{b128}, \text{hull}_{d64}, \text{hull}_{d128}.
\]

8.8.2. Interval conversion. An implementation shall provide, for each supported bare interval type \(T\), an operation that returns \(\text{hull}_T(x)\) (as an interval of type \(T\)) for any interval \(x\) of any supported bare interval type.

8.8.3. Hull in several dimensions. In \(n\) dimensions the \(T\)-hull is defined componentwise, namely for an arbitrary subset \(s\) of \(\mathbb{R}^n\) it is \(\text{hull}_T(s) = (y_1, \ldots, y_n)\) where

\[
y_i = \text{hull}_T(s_i),
\]

and \(s_i = \{ s_i \mid s \in s \}\) is the projection of \(s\) on the \(i\)th coordinate dimension. It is easily seen that this is a minimal \(T\)-box containing \(s\), and that if \(T\) is explicit it equals the unique tightest \(T\)-box containing \(s\).

8.9. Level 2 interval extensions. Let \(T\) be a bare interval type and \(f\) an \(n\)-variable scalar point function. A \(T\)-interval extension of \(f\) is a mapping \(f\) from \(n\)-dimensional \(T\)-boxes to \(T\)-intervals, that is \(f : T^n \to T\), such that \(f(x) \in f(x)\) whenever \(x \in x\) and \(f(x)\) is defined.

Equivalently

\[
f(x) \supseteq \text{Range}(f \mid x).
\]

for any \(T\)-box \(x \in T^n\), regarding \(x\) as a subset of \(\mathbb{R}^n\). Generically, such mappings are called Level 2 interval extensions.

Though only defined over a finite set of boxes, a Level 2 extension of \(f\) is equivalent to a full Level 1 extension of \(f\) (§7.4.3) so that this document does not distinguish between Level 2 and Level 1 extensions. Namely define \(f^*\) by

\[
f^*(s) := f(\text{hull}_T(s))
\]

for any subset \(s\) of \(\mathbb{R}^n\). Then the interval \(f^*(s)\) contains \(\text{Range}(f \mid s)\) for any \(s\), making \(f^*\) a Level 1 extension, and \(f^*(s)\) equals \(f(s)\) whenever \(s\) is a \(T\)-box.

8.10. Accuracy modes for inf-sup types. The standard defines accuracy modes that indicate how near an operation is to being as tight as possible.

[Note. These modes are specified for inf-sup types only. “Tightest” and “valid” apply to any interval type, but there is no simple way to define an analogue of “accurate” for general types.]

For a given number format \(F\) and an extended-real number \(x\), \(\text{nextUp}(x)\) is defined to be \(+\infty\) if \(x = +\infty\), and the least member of \(F\) greater than \(x\) otherwise; \(\text{nextDown}(x)\) is defined to be \(-\infty\) if \(x = -\infty\), and the greatest member of \(F\) less than \(x\) otherwise.

Given an interval \(x = [\underline{x}, \overline{x}]\) of the inf-sup type \(T\) derived from \(F\), \(\text{widen}(x)\) is the \(T\)-interval defined by

\[
\text{widen}(x) = [\text{nextDown}(x), \text{nextUp}(x)].
\]

[Note. That is, \(\text{widen}\) moves each finite endpoint of \(x\) outward to the next \mathbb{F}-number. For 754 formats and others based on a radix-significand-exponent form, this is often called a change of one ulp (unit in the last place).]

For a \(T\)-box \(x = (x_1, \ldots, x_n)\), this function acts componentwise to produce the \(T\)-box

\[
\text{widen}(x) = (\text{widen}(x_1), \ldots, \text{widen}(x_n)).
\]

For a \(T\)-interval extension \(f\) of an \(n\)-variable scalar point function \(f\), the standard specifies three accuracy modes for \(f\):

Tightest: \(f(x)\) shall equal \(\text{hull}_T(\text{Range}(f \mid x))\), for any \(T\)-box \(x\).
§8.11

Accurate: \( f(x) \) shall be contained in \( \text{widen}(\text{hull}_T(\text{Range}(f \mid \text{widen}(x)))) \), for any \( T \)-box \( x \). That is, the result lies within a slightly expanded hull of the exact range of a slightly expanded input box.

Valid: No requirement beyond \[24\].

[Note. In the “accurate” specification, the second \( \text{widen}() \) aims to handle the problem of a function like \( \sin x \) evaluated at a very large argument, where a small relative change in the input can produce a large relative change in the result. The first \( \text{widen}() \) is to handle the problem of correct rounding, which may be hard or even undecidable for some special functions at some arguments. ]

8.11. Required operations on bare intervals.

An implementation shall provide a \( T \)-version \( (§8.1) \) of each operation listed in subclauses \( §8.11.1 \) and \( §8.11.2 \) to \( §8.11.9 \), for each supported type \( T \). That is, those of its inputs and outputs that are intervals, are of type \( T \).

A 754-conforming implementation, or part thereof, shall provide mixed-type \( \text{typeOf} \) operations, as specified in \( §8.3.1 \), for the following operations, which correspond to those that 754 requires to be provided as \( \text{formatOf} \) operations.

\[ \text{add, sub, mul, div, inv, sqrt, sqr, sign, ceil, floor, round, trunc, abs, min, max, fma} \]

An implementation may provide more than one version of some operations for a given type. For instance it may provide an “accurate” version of some operation in addition to a required “tightest” one, to offer a trade-off between accuracy and speed. How such a facility is provided, is language- or implementation-defined.

8.11.1. Interval literals.

An interval literal is a text string denoting an interval. At Level 2 it is defined to be any string conforming to the interchange syntax, or an implementation-defined extended syntax, as described below. Such a string is also called valid and any other string is invalid (as interval literals).

Implementation-defined are: the character set; the character encoding; and possible locale-dependent variations.

At both Level 1 and Level 2, a valid string denotes an exact mathematical interval, its value, as specified below. When an implementation converts it to an interval of some supported type \( T \), this interval shall in all cases enclose the mathematical interval.

An invalid string is deemed to have the value Empty at Level 2.

The primary means of conversion is by calling a \( T \)-version of the function \( \text{text2interval} \), but languages may define other contexts where interval literals are converted to internal intervals in a program.

A numeric floating literal (cf. strings accepted by the C function \( \text{strtod} \)) is one of:

- A decimal or hexadecimal floating constant as defined in the C standard [WG14 N1256 (2007), §6.4.4.2]. Its value is the exact real number it represents.
- The string \( \text{inf} \) or \( \text{infinity} \), ignoring case, optionally preceded by \( + \), with value \( +\infty \); or preceded by \( - \), with value \( -\infty \).

A string conforming to the interchange syntax has one of the following forms, where the tokens (lexical elements) are optionally separated by one or more space characters. To simplify stating the needed constraints, e.g. \( l \leq u \), the literals \( l, u, m, r \) are identified with the numbers they represent.

(i) An inf-sup string, of the form \( [l \; u] \) where \( l \) and \( u \) are numeric floating literals with \( l \leq u \), \( l < +\infty \) and \( u > -\infty \), see \( §7.2 \). Its value is the mathematical interval \( [l, u] \).
(ii) A mid-rad string, of the form \( <m \; r> \) where \( m \) and \( r \) are numeric floating literals representing finite numbers with \( r \geq 0 \). Its value is the mathematical interval \( [m - r, m + r] \).
(iii) The strings \( \text{empty} \), ignoring case, whose value is the empty set \( \emptyset \); and \( \text{entire} \), ignoring case, whose value is the whole line \( \mathbb{R} \).

A formal description of strings accepted by the interchange syntax is by the following grammar (using the notation of 754§5.12.3) which defines an \( \text{intervalLiteral} \), subject to the constraints on
l, u, m, r stated above.

floatLiteral  { numeric floating literal in the default locale }

sp             { space character } *

infSupInterval "[" { floatLiteral } { sp } "," { sp } { floatLiteral } { sp } "]*

midRadInterval ">" { sp } { floatLiteral } { sp } "+-" { sp } { floatLiteral } { sp } ">" 

intervalLiteral ( { infSupInterval } | { midRadInterval } | "empty" | "entire" ) 

In an extended syntax, which is language- or implementation-defined:

- sp should be allowed to be a sequence of zero or more language-defined white-space characters.
- Other ways of denoting an interval (besides inf-sup and mid-rad) may be provided.
- Locale-dependent variations may be provided.
- Floating literals may include denotations of real constants such as \( \pi \)
- The inf-sup style \([x]\) should be provided, with the same meaning as \([x,x]\).

[Example. With these extensions, for any type \( T \), applying the \( T \) version of text2interval to the string \( \text{Pi} \) gives a \( T \)-interval enclosing \( \pi \).]

8.11.2. Forward-mode elementary functions. An implementation shall provide a \( T \)-version of each forward arithmetic operation in §7.6.4 for each supported bare interval type \( T \). Its inputs and output are \( T \)-intervals.

For a 754-conforming type, each such operation shall have a version of that type with accuracy mode as in Table 9. For other types, the accuracy mode is language- or implementation-defined.

[Note. For operations with some integer arguments, such as integer power \( x^n \), only the real arguments are replaced by intervals.]

8.11.3. Interval case expressions and case function. An implementation shall provide the interval case \( (c, g, h) \) function, see §7.6.5 for each supported type \( T \). The input \( c \) is of an arbitrary supported interval type. The inputs \( g, h \), and the result, are of type \( T \). The implementation shall be as if the \( T \)-version of \( \text{convexHull} \) is used in (6) of §7.6.3.

8.11.4. Reverse-mode elementary functions. An implementation shall provide a \( T \)-version of each reverse arithmetic operation in §7.6.4 for each supported bare interval type \( T \). Its inputs and output are \( T \)-intervals.

For a 754-conforming type, each such operation shall have a version of that type with accuracy mode as in Table 9. For other types, the accuracy mode is language- or implementation-defined.

8.11.5. Cancellative addition and subtraction. An implementation shall provide a \( T \)-version of each of the operations cancelPlus and cancelMinus in §7.6.5 for each supported bare interval type \( T \). Its inputs and output are \( T \)-intervals.

It shall return an enclosure of the Level 1 result if the latter is defined. In particular it shall return Empty if the Level 1 value is Empty. Otherwise it shall return Entire. Thus, for the case of cancelMinus(\( x, y \)), it returns Entire in these cases:

- both \( x \) and \( y \) are unbounded;
- \( x \neq \emptyset \) and \( y = \emptyset \);
- \( x \) and \( y \) are nonempty bounded intervals with width(\( x \)) \(<\) width(\( y \)).

It returns an unbounded interval, which may be Entire, if the Level 1 value is defined but there is no bounded \( T \)-interval containing it.

If \( T \) is a 754-conforming type, the result shall be the \( T \)-hull of the Level 1 result when this is defined. [Note. This may require computation in extra precision in boundary cases: see the example in §7.6.]

Implementations should only provide the operations of this subclause in decorated form, to make “no value at Level 1” detectable.

8.11.6. Non-arithmetic operations. An implementation shall provide a \( T \)-version of each of the operations intersection and convexHull in §7.6.6 for each supported bare interval type \( T \). Its inputs and output are \( T \)-intervals.

These operations shall return the \( T \)-interval hull of the exact result.

[Note. In particular, if \( T \) is an inf-sup type, each operation always returns the exact result. However, this need not be the case with the mixed-type version of the operation, when \( T \) is a 754-conforming type.]
8.11.7. Constructors.

There shall be a bare interval version of each constructor in §7.6.7, for each supported bare interval type $T$. It returns a $T$-interval. However, it is recommended that implementations should not provide the bare interval constructors `nums2interval` and `text2interval` at user level. Only the decorated interval form should be provided to users.

Both `nums2interval` and the inf-sup form of `text2interval` involve testing if $b = (l \leq u)$ is 0 (false) or 1 (true), to determine whether the interval is empty or nonempty. In the former case, $l$ and $u$ are values of supported number formats within a program; in the latter, they are floating literals. An implementation must never evaluate $b$ as 0 when the true value is 1 (a “false negative”), since this leads to falsely returning Empty as an enclosure of the true nonempty interval.

Evaluating $b$ correctly can be hard, if $l$ and $u$ have values very close in a relative sense, and are represented in different ways—e.g., if an implementation allows them to be floating point variables or literals of different radices. It could be especially challenging in an extendable-precision context.

Language rules can cause such errors even when $l$ and $u$ have the same format. E.g., in C, if `long double` is supported and has more precision than `double`, default behavior might be to round `long double` inputs $l$ and $u$ to `double` at entry to a `nums2interval` call. This is

<table>
<thead>
<tr>
<th>Name</th>
<th>Accuracy</th>
<th>Name</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(x, y)</code></td>
<td>tightest</td>
<td><code>sqrRev(c, x)</code></td>
<td>accurate</td>
</tr>
<tr>
<td><code>sub(x, y)</code></td>
<td>tightest</td>
<td><code>invRev(c, x)</code></td>
<td>accurate</td>
</tr>
<tr>
<td><code>mul(x, y)</code></td>
<td>tightest</td>
<td><code>absRev(c, x)</code></td>
<td>accurate</td>
</tr>
<tr>
<td><code>div(x, y)</code></td>
<td>tightest</td>
<td><code>pownRev(c, x, p)</code></td>
<td>accurate</td>
</tr>
<tr>
<td><code>inv(x)</code></td>
<td>tightest</td>
<td><code>sinRev(c, x)</code></td>
<td>accurate</td>
</tr>
<tr>
<td><code>sqrt(x)</code></td>
<td>tightest</td>
<td><code>cosRev(c, x)</code></td>
<td>accurate</td>
</tr>
<tr>
<td><code>hypot(x, y)</code></td>
<td>tightest</td>
<td><code>tanRev(c, x)</code></td>
<td>accurate</td>
</tr>
<tr>
<td><code>case(b, g, h)</code></td>
<td>tightest</td>
<td><code>atan2Rev1(b, c, x)</code></td>
<td>accurate</td>
</tr>
<tr>
<td><code>sqrt(x)</code></td>
<td>tightest</td>
<td><code>atan2Rev2(a, c, x)</code></td>
<td>accurate</td>
</tr>
<tr>
<td><code>pown(x, p)</code></td>
<td>accurate</td>
<td><code>tan(x)</code></td>
<td>accurate</td>
</tr>
<tr>
<td><code>pow(x, y)</code></td>
<td>accurate</td>
<td><code>cosh(x)</code></td>
<td>accurate</td>
</tr>
<tr>
<td><code>exp,exp2,exp10(x)</code></td>
<td>tightest</td>
<td><code>acosh(x)</code></td>
<td>accurate</td>
</tr>
<tr>
<td><code>log,log2,log10(x)</code></td>
<td>tightest</td>
<td><code>atanh(x)</code></td>
<td>accurate</td>
</tr>
<tr>
<td><code>sin(x)</code></td>
<td>accurate</td>
<td><code>sign(x)</code></td>
<td>tightest</td>
</tr>
<tr>
<td><code>cos(x)</code></td>
<td>accurate</td>
<td><code>cei(x)</code></td>
<td>tightest</td>
</tr>
<tr>
<td><code>tan(x)</code></td>
<td>accurate</td>
<td><code>floor(x)</code></td>
<td>tightest</td>
</tr>
<tr>
<td><code>asin(x)</code></td>
<td>accurate</td>
<td><code>round(x)</code></td>
<td>tightest</td>
</tr>
<tr>
<td><code>acos(x)</code></td>
<td>accurate</td>
<td><code>trunc(x)</code></td>
<td>tightest</td>
</tr>
<tr>
<td><code>atan(x)</code></td>
<td>accurate</td>
<td><code>abs(x)</code></td>
<td>tightest</td>
</tr>
<tr>
<td><code>atan2(y, x)</code></td>
<td>accurate</td>
<td><code>min(x_1, ..., x_k)</code></td>
<td>tightest</td>
</tr>
<tr>
<td><code>asinh(x)</code></td>
<td>accurate</td>
<td><code>max(x_1, ..., x_k)</code></td>
<td>tightest</td>
</tr>
</tbody>
</table>

Table 9. Accuracy levels for required arithmetic operations.
forbidden—the comparison \( l \leq u \) requires the exact values to be used, which requires use of a version of \texttt{nums2interval} with \texttt{long double} arguments.

Evaluating \( b \) as 1 when the true value is 0 (a “false positive”) is undesirable, but permissible since it returns a nonempty interval as an enclosure for Empty. Implementations shall ensure that false negatives cannot occur, and should ensure that false positives cannot occur.

\texttt{nums2interval}.

- The inputs \( l \) and \( u \) to the constructor \( \mathbf{x} = \texttt{nums2interval}(l, u) \) are datums of supported number formats \( F_l \) and \( F_u \). Mixed format, where \( F_l \neq F_u \), is possible. For a given \( T \), each \( F_l, F_u \) combination is called a kind in this subclause.

For all kinds, the result \( \mathbf{x} \) shall enclose the Level 1 value if this exists, that is, if neither \( l \) nor \( u \) is NaN, and the exact extended-real values of \( l \) and \( u \) satisfy \( l \leq u \), \( l < +\infty \), \( u > -\infty \).

The cases \( l = u = -\infty \) or \( l = u = +\infty \) are special, see below.

Otherwise \( \mathbf{x} \) should be Empty. Rarely, this may not be possible if \( l > u \) but the implementation cannot determine that this is so, see the discussion at the start of this subclause.

- An implementation shall provide at least one kind where \( l \) and \( u \) have the same format. This format should be compatible with \( T \), see §8.5.

- For \( T \) belonging to a 754-conforming implementation (or part implementation), \texttt{formatOf} kinds shall be provided, which accept \( l \) and \( u \) having any 754 format of that implementation (or part), of the same radix as \( T \)’s parent format. The result shall be the \( T \)-hull of the Level 1 result, when this exists, and shall be Empty otherwise.

- If \( l = u = +\infty \), the Level 1 result is undefined. In finite precision however, \( l \) and \( u \) are likely to be finite values that overflew. Therefore in this case \texttt{nums2interval} shall return the tightest \( T \)-interval that is unbounded above. For any type, this exists and has the form \( \mathbf{x} = \texttt{[HUGEPOS, +\infty]} \), where \texttt{HUGEPOS} is a uniquely defined extended-real number \( < +\infty \).

[Note. If \( T \) is an inf-sup type based on a format \( \mathbb{F} \), then \texttt{HUGEPOS} is the largest finite \( \mathbb{F} \)-number. \texttt{HUGEPOS} can be \( -\infty \), hence \( \mathbf{x} = \text{Entire} \), if no \( T \)-intervals are unbounded above, except \text{Entire}.]

If \( l = u = -\infty \), \texttt{nums2interval} shall return \([-\infty, \texttt{HUGENEG}]\), defined similarly.

\texttt{text2interval}.

Input \( s \) to the constructor \( \texttt{text2interval}(s) \) is a text string. If \( s \) is a valid interval literal with value \( \mathbf{x} \), see §8.11.1 the result shall be a \( T \)-interval containing \( \mathbf{x} \); otherwise the result is Empty.

If \( T \) is a 754-conforming type, the result shall be the \( T \)-hull of \( \mathbf{x} \).

\texttt{bareempty, bareentire}.

The \( T \)-versions of the constructors \texttt{bareempty()} and \texttt{bareentire()} return the \( T \)-versions of \texttt{Empty} and \texttt{Entire} respectively.

8.11.8. \textit{Numeric functions of intervals.} An implementation shall provide a \( T \)-version of each numeric function in Table 3 of §7.6.8 for each supported bare interval type \( T \). It shall return a result in a supported number format \( \mathbb{F} \) as defined in §8.5. Several, user-selectable, versions may be provided, returning results in different formats.

The implementation shall document how it breaks ties, e.g., when computing the closest \( \mathbb{F} \)-number to a value that is midway between two \( \mathbb{F} \)-numbers.

If \( T \) is a 754-conforming type, versions shall be provided that return the result in any supported 754 format of the same radix as \( T \).

- \( \text{inf}(\mathbf{x}) \) returns the Level 1 value, rounded toward \( -\infty \).
- \( \text{sup}(\mathbf{x}) \) returns the Level 1 value, rounded toward \( +\infty \).
- \( \text{mid}(\mathbf{x}) \) returns the closest \( \mathbb{F} \)-number to the Level 1 value if the latter exists. Otherwise it returns NaN.
- \( \text{rad}(\mathbf{x}) \) returns NaN if the Level 1 value does not exist (if \( \mathbf{x} \) is empty) and \( +\infty \) if \( \mathbf{x} \) is unbounded. Otherwise it returns the smallest \( \mathbb{F} \)-number \( r \) such that \( \mathbf{x} \) is contained in the exact interval \([m - r, m + r]\), where \( m \) is the value returned by \( \text{mid}(\mathbf{x}) \).

[Note. \( \text{rad}(\mathbf{x}) \) may be \( +\infty \) even though \( \mathbf{x} \) is bounded, if \( \mathbb{F} \) has insufficient range. However, if \( \mathbb{F} \) is a 754 format and \( T \) is the derived inf-sup type, \( \text{rad}(\mathbf{x}) \) is finite for all bounded nonempty intervals.]

- \( \text{wid}(\mathbf{x}) \) returns the same value as \( 2 \cdot \text{rad}(\mathbf{x}) \), rounded toward \( +\infty \).

[Note. At Level 2, \( \text{wid}(\mathbf{x}) \) may be infinite though \( \text{rad}(\mathbf{x}) \) is finite.]

- \( \text{mag}(\mathbf{x}) \) returns the Level 1 value rounded toward \( +\infty \) if this value exists (if \( \mathbf{x} \) is nonempty). Otherwise it returns NaN.
8.1.3. Decorated interval types.

To come shortly.

8.12. Recommended operations (informative).

To come shortly.