

P1788/D7.0

Draft Standard For Interval Arithmetic

John Pryce and Christian Keil, Technical Editors

DRAFT 7.0

DRAFT 7.0

Frontmatter

Required IEEE items to be added:

- Draft copyright statements
- Title
- Abstract and keywords
- Committee lists
- Acknowledgments (after the Introduction)

⚠ To the P1788 working group 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.

Introduction

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.

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.

This document uses the standard notation $[a, b]$ for “the interval between numbers a and b ”, with various detailed meanings depending on the underlying theory. The “classical” interval arithmetic (IA) of R.A. Moore [5] uses only bounded, closed, nonempty intervals in the real numbers \mathbb{R} —that is, $[a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$ where $a, b \in \mathbb{R}$ with $a \leq b$. So, for instance, division by an interval containing 0 is not defined in it. It was agreed early on that this standard should strictly extend classical IA in virtue of allowing an interval to be unbounded or empty.

Beyond this, various extensions of classical IA were considered. One choice that distinguishes between theories is: Are arithmetic operations purely algebraic, or do they involve topology? An example of the latter is containment set (cset) theory [8], which extends functions over the reals to functions over the extended reals, e.g. $\sin(+\infty)$ is the set of all possible limits of $\sin x$ as $x \rightarrow +\infty$, which is $[-1, 1]$. The complications of this were deemed to outweigh the advantages, and it was agreed that operations should be purely algebraic.

Another choice is: Is an interval a set—a subset of the number line—or is it something different? The most widely used forms of IA are *set-based* and define an interval to be a set of real numbers. They have established software to find validated solutions of linear and nonlinear algebraic equations, optimization problems, differential equations, etc.

However *Kaucher* IA and the nearly equivalent *modal* IA have significant applications. In the former an interval is formally a pair (a, b) of real numbers, which for $a \leq b$ is “proper” and identified with the normal interval $\{x \in \mathbb{R} \mid a \leq x \leq b\}$, and for $a > b$ is “improper”. In the latter, an interval is a pair (X, Q) where X is a normal interval and Q is a quantifier, either \exists or \forall . At the time of writing it finds commercial use in the graphics rendering industry. Both forms are referred to as Kaucher IA henceforth.

In view of their significance it was decided to support both set-based and Kaucher IA. Because of their different mathematical bases this led to the concept of *flavors* (see Clause 5). 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, e.g., division by an interval containing zero), all flavors produce the same results, at the mathematical level and also—up to roundoff—in finite precision.

Currently the standard incorporates two flavors, set-based and Kaucher. Others could be added, though there are no current plans to do so. E.g., csets could be a flavor, since they also extend classical IA in the defined sense.

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 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. At the mathematical level (Level 1 in the structure defined in §4.1) they are precisely all topologically closed and connected subsets of \mathbb{R} . The finite-precision level (Level 2), uses the notion of an interval type, which is a finite set of Level 1 intervals.
- The interval version of an elementary function such as $\sin x$ is essentially the natural algebraic extension to sets of the corresponding pointwise function on real numbers.

Fuzzy sets, like intervals, are a way to handle uncertain knowledge, and the two topics are related. However, considering this relation was beyond the scope of this project.

Specification Levels. The 754-2008 standard describes itself as layered into four Specification Levels. To manage complexity, the present standard 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. It was a recognized omission of IEEE-754-1985 that it specified individual operations but not how they should be used in expressions. Optimizing compilers have, since well before that standard, used clever transformations so that it is impossible to know the precisions used and the roundings performed while evaluating an expression, or whether the compiler has even “optimized away” $(1.0 + x) - 1.0$ to become simply x . IEEE-754-2008 specifies this by placing requirements on how operations should be used in expressions, though as of this writing, few programming languages have adopted that.

The lack of any restrictions 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 most current languages sit 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). This standard extends them to provide an instantiation of level 2 entities.

The Fundamental Theorem. Moore’s [5] 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 , \dots , with no non-arithmetic operations such as intersection*, so that it defines a real function $f(x_1, \dots, x_n)$. Then evaluating f “in interval mode” over any interval inputs (x_1, \dots, 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 achievements were 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.

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, report an illegal argument 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 an interval enclosing 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. A language might achieve this by binding one of the above three interpretations—usually some variant of (2)—to its built-in operations, and providing the others as library procedures.

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.

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 continuous on a box in \mathbb{R}^n defined by n input intervals x_1, \dots, 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. In this standard, such events are recorded locally by *decorations*.

A *decorated interval* is an ordinary interval tagged with a few bits that encode the decoration, and record while evaluating an expression, e.g., “each elementary function was defined and continuous on its inputs”—which implies the same for the function defined by the whole expression. This makes possible a rigorous check of properties such as listed in item (2) of §. A small number of decorations is provided, designed for efficient propagation of such property information.

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. With large problems on typical machine architectures this may cause inefficiencies—in data throughput if storing 17-byte data structures, or in storage if one pads the structure to, say, 32 bytes. Hence an optional *compressed* decorated interval scheme is specified, 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.

DRAFT 7.0

Contents

Frontmatter	iii
Introduction	iii
Mathematical context	iii
Specification Levels	iv
The Fundamental Theorem	iv
Operations	v
Decorations	v
 Chapter 1. General Requirements	 1
1. Overview	1
1.1. Scope	1
1.2. Purpose	1
1.3. Inclusions	1
1.4. Exclusions	1
1.5. Word usage	1
1.6. The meaning of conformance	2
1.7. Programming environment considerations	2
1.8. Language considerations	2
2. Normative references	3
3. Notation, abbreviations, definitions	4
3.1. Frequently used notation and abbreviations	4
3.2. Definitions	4
4. Structure of the standard in levels	6
4.1. Specification levels overview	6
5. Flavors	7
5.1. Flavors overview	7
5.2. Definition of common intervals and common evaluations	7
5.3. Loose common evaluations	8
5.4. Relation of common evaluations to flavors	8
5.5. Flavors and the Fundamental Theorem	9
6. Expressions and the functions they define	9
7. Decoration system	11
7.1. Decorations overview	11
7.2. Decoration definition and propagation	12
7.3. Recognizing common evaluation	12
8. Conformance requirements	13
 Chapter 2. Set-Based Intervals	 15
9. Level 1 description	15
9.1. Non-interval Level 1 entities	15
9.2. Intervals	15
9.3. Hull	16
9.4. Functions	16
9.4.1. Function terminology	16
9.4.2. Point functions	16
9.4.3. Interval-valued functions	16
9.4.4. Constants	17

9.5. Expressions	17
9.6. Required operations	18
9.6.1. Interval literals	18
9.6.2. Interval constants	18
9.6.3. Forward-mode elementary functions	18
9.6.4. Interval case expressions and case function	18
9.6.5. Reverse-mode elementary functions	20
9.6.6. Cancellative addition and subtraction	21
9.6.7. Non-arithmetic (set) operations	22
9.6.8. Constructors	22
9.6.9. Numeric functions of intervals	22
9.6.10. Boolean functions of intervals	22
9.6.11. Dot product function	23
9.7. Recommended operations	24
9.7.1. Forward-mode elementary functions	24
9.7.2. Extended interval comparisons	24
9.7.3. Slope functions	25
10. The decoration system at Level 1	28
10.1. Decorations and decorated intervals overview	28
10.2. Definitions and basic properties	28
10.3. Ill-formed intervals	29
10.4. Permitted combinations	29
10.5. Initial decoration	29
10.6. Decorations and arithmetic operations	30
10.7. Non-arithmetic operations	31
10.8. Disassembling and assembling decorated intervals	31
10.9. User-supplied functions	31
10.10. The <code>com</code> decoration	32
10.11. Compressed arithmetic with a threshold	33
11. Level 2 description	36
11.1. Level 2 introduction	36
11.2. Naming conventions for operations	36
11.3. 754-conformance	36
11.3.1. 754-conforming mixed-type arithmetic	37
11.4. Tagging, and the meaning of equality at Level 2	37
11.5. Number formats	38
11.6. Bare interval types	38
11.6.1. Definition	38
11.6.2. Inf-sup and mid-rad types	39
11.7. Multi-precision interval types	39
11.8. Explicit and implicit types, and Level 2 hull operation	39
11.8.1. Hull in one dimension	39
11.8.2. Hull in several dimensions	40
11.8.3. Interval type conversion	40
11.9. Level 2 interval extensions	40
11.10. Accuracy modes for inf-sup types	40
11.11. Required operations on bare intervals	41
11.11.1. Interval literals	41
11.11.2. Interval constants	42
11.11.3. Forward-mode elementary functions	42
11.11.4. Interval case expressions and case function	42
11.11.5. Reverse-mode elementary functions	42
11.11.6. Cancellative addition and subtraction	42
11.11.7. Set operations	42
11.11.8. Constructors	42
11.11.9. Numeric functions of intervals	44

11.11.10.	Boolean functions of intervals	45
11.11.11.	Complete arithmetic, dot product function	45
11.12.	Recommended operations	45
12.	The decoration system at Level 2	45
12.1.	Decorated interval types	45
12.2.	Required decorated types	46
12.3.	Decorated versions of an operation	46
12.4.	Required operations on decorated intervals	46
12.4.1.	Interval literals	46
12.4.2.	Interval constants	46
12.4.3.	Forward-mode elementary functions	46
12.4.4.	Interval case expressions and case function	46
12.4.5.	Interval-valued non-arithmetic operations	46
12.4.6.	Constructors	46
12.4.7.	Numeric functions of intervals	47
12.4.8.	Boolean functions of intervals	47
12.5.	Compressed arithmetic at Level 2	47
13.	Input and output (I/O) of intervals	48
13.1.	Overview	48
13.2.	Input	48
13.3.	Output	48
13.4.	Public representation	49
14.	Level 3 description	50
14.1.	Representation of intervals by lower/upper bounds	50
14.2.	Format conversion	50
14.3.	Interchange formats	50
14.4.	Support levels for interval elementary functions	50
14.5.	Operation tables for basic interval operations	51
14.6.	Care needed with innerPlus and innerMinus	53
14.7.	Implementation of bare object arithmetic	53
15.	Level 4 description	54
Chapter 3.	Kaucher Intervals	55
Annex A.	Details of flavor-independent requirements	57
16.	List of required functions	57
17.	List of recommended functions	57
Annex B.	Including a new flavor in the standard	59
Annex C.	Set-based decorations: details and examples	61
18.	Local decorations of arithmetic operations	61
18.1.	Forward-mode elementary functions	61
18.2.	Interval case function	61
19.	Examples of use of decorations	61
20.	Decoration system: rigorous theory and proofs	65
20.1.	The fundamental theorem of decorated interval arithmetic	65
20.2.	Compressed interval proofs	66
Annex D.	Further material (informative)	67
21.	Type conversion in mixed operations	67
22.	The “Not an Interval” object	67
23.	Implementation of compressed interval arithmetic (informative)	69
Bibliography		71

DRAFT 7.0

General Requirements

1. Overview

1.1. Scope. This standard specifies basic interval arithmetic (IA) operations selecting and following one of the commonly used mathematical interval models. This standard supports the IEEE-754-2008 floating point formats of practical use in interval computations. Exception conditions are defined and standard handling of these conditions is 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; lack of a standard inhibits development, portability; ability to verify correctness of codes.

1.3. Inclusions. This standard specifies

- Types for interval data based on underlying numeric formats, with a special class of type derived from IEEE 754 floating point formats.
- Constructors for intervals from numeric and character sequence data.
- Addition, subtraction, multiplication, division, fused multiply add, square root; other interval-valued operations for intervals.
- Midpoint, radius and other numeric functions of intervals.
- Interval comparison relations.
- Required elementary functions.
- 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 numeric formats supported by the underlying system shall have an associated interval type.
- Details of how an implementation represents intervals at the level of programming language data types, or bit patterns.

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;
- **comprise** indicates 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 format consists of numbers with a terminating binary representation”. “Comprises” means “consists exactly of”.
- **Note** and **Example** introduce text that is informative (is not a requirement of this standard).

1.6. The meaning of conformance. Clause 8 lists the requirements on a conforming implementation in summary form, with references to where these are stated in detail.

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; 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. 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.

1.8. Language considerations. All relevant languages are based on the concepts of data and transformations. In Von Neumann languages, data are held in variables, which are transformed by assignment. In functional languages, input data are supplied as arguments; the transformed form is returned as results. Dataflow languages vary considerably, but use some form of the data and transformation approach.

Similarly, all relevant languages are based on the concept of mapping the pseudo-mathematical notation that is the program code to approximate real arithmetic, nowadays almost exclusively using some form of floating-point. The unit of mapping and transformation can be individual operations and built-in functions, expressions, statements, complete procedures, or other. This standard is applicable to all of these.

The least requirement on a conforming language standard, compiler or interpreter is that it shall:

- (1) define bindings so that the programmer can specify level 2 data (in the sense of the levels defined in §4.1) as described in this standard;
- (2) define bindings so that the programmer can specify the operations on such data as described in this standard;
- (3) define any properties of such data and operations that this standard requires to be defined;
- (4) honor the rules of interval transformations on such data and operations as described in this standard; such units of transformation that the language standard, compiler or interpreter uses.

Specifically, if the data before and after the unit of transformation are regarded as sets of mathematical intervals, the transformed form of all combinations of the elements (the real values) represented by the prior set shall be a member of the posterior set.

If a conforming language standard supports reproducible interval arithmetic it shall also:

- (5) Use the data bindings as specified in point (1) above for reproducible operations;
- (6) Define bindings to the reproducible operations as described in this standard;
- (7) Define any modes and constraints that the programmer needs to specify or obey in order to obtain reproducible results.

If a conforming language standard supports both non-reproducible and reproducible interval arithmetic it shall also:

- (8) Permit a reproducible transformation unit to be used as a component in a non-reproducible program, possibly via a suitable wrapping interface.

2. Normative references

1. IEEE Std 754-2008. IEEE Standard for Floating-Point Arithmetic.

DRAFT 7.0

3. Notation, abbreviations, definitions

3.1. Frequently used notation and abbreviations.

754	IEEE-Std-754-2008 “IEEE Standard for Floating-Point Arithmetic”.
\mathbb{R}	the set of real numbers.
$\overline{\mathbb{R}}$	the set of extended real numbers, $\mathbb{R} \cup \{-\infty, +\infty\}$.
\mathbb{IR}	the set of closed real intervals, including unbounded intervals and the empty set.
$\mathbb{F}, \mathbb{G}, \dots$	generic notation for (the set of numbers, including $\pm\infty$) representable in some floating point format.
$\mathbb{IF}, \mathbb{IG}, \dots$	the members of \mathbb{IR} whose lower and upper bounds are in $\mathbb{F}, \mathbb{G}, \dots$
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, \dots [resp. f, g, \dots]	typeface/notation for a numeric value [resp. numeric function].
$\mathbf{x}, \mathbf{y}, \dots$ [resp. $\mathbf{f}, \mathbf{g}, \dots$]	typeface/notation for an interval value [resp. interval function].
f, g, \dots	typeface/notation for an expression, producing a function by evaluation.
$\text{Dom}(f)$	the domain of a point-function f .
$\text{Rge}(f \mid \mathbf{s})$	the range of a point-function f over a set \mathbf{s} ; the same as the image of \mathbf{s} under f .

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

3.2. Definitions.

 Definitions belonging to Levels 2 onward have been temporarily removed.

3.2.1. arithmetic operation. A function provided by an implementation (see Defn 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 **(bare) interval extensions** of the point operation, each of which corresponds to the finite precision interval type of its result; and one or more **decorated interval extensions**, each being the (unique) decorated version of a bare interval extension.

Together with the interval non-arithmetic operations (§9.4.1), these form the implementation’s **library**, which splits into the **point library** (a conceptual entity, being a set of mathematical functions), the **bare interval library** and the **decorated interval library**, corresponding to the above categories. The latter two may be further qualified by a result interval type, e.g., “binary64 inf-sup decorated interval library”.

The programming environment’s floating point approximations to mathematical point functions constitute the *floating point library*. The standard makes no requirements on these.

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

Constants such as 3.456 and π are regarded as arithmetic operations whose number of arguments is zero. Details in §9.4.4.

3.2.2. box. See Defn 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 Defn 3.2.14.

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 $\varphi(g_1, \dots, g_k)$, where φ 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

operations. Writing $f(z_1, \dots, z_n)$ 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 .

Details in §9.5. For the ways in which an expression defines a function, see §??, ??.

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, \dots, x_n) where the x_i are intervals. Often identified with the cartesian product $x_1 \times \dots \times x_n \subseteq \mathbb{R}^n$, it is empty if any of the x_i is empty. Details in §9.2.

3.2.11. **interval extension.** An interval extension of a point function f is a function \mathbf{f} from intervals to intervals such that $f(x)$ belongs to $\mathbf{f}(x)$ whenever x belongs to x and $f(x)$ is defined. Details in §9.4.3.

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 §9.4.3.

3.2.13. **interval library.** See Defn 3.2.1.

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

3.2.15. **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.

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

3.2.17. **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 §9.1.

3.2.18. **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 $f(z_1, \dots, z_n)$ defines a (usually scalar) point function, whose domain is the natural domain of f .

3.2.19. **point library.** See Defn 3.2.1.

3.2.20. **range.** The range, $\text{Rge}(f | s)$, 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. $\{f(x) \mid x \in s \text{ and } x \in \text{Dom } f\}$.

3.2.21. **string.** A text string, or just string, is a finite sequence of characters belonging to some alphabet. See §9.1.

3.2.22. **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.

3.2.23. **version.** For brevity, bare or decorated interval extensions, or floating point approximations, of a library point operation may be called versions of that operation.

Relationships between specification levels for interval arithmetic for a given flavor \mathfrak{F} and a given finite-precision interval type \mathbb{T} .		
Level 1	Number system used by flavor \mathfrak{F} . Set of allowed intervals in \mathfrak{F} . Principles of how $+$, $-$, \times , \div and other arithmetic operations are extended to intervals.	Mathematical Model level.
	$\downarrow \mathbb{T}$ -interval hull total, many-to-one ^a	<i>identity map</i> \uparrow total, one-to-one ^b
Level 2	A finite subset \mathbb{T} of the \mathfrak{F} -intervals—the \mathbb{T} -interval datums—and operations on them.	Interval datum level.
		<i>“represents”</i> \uparrow partial, many-to-one, onto ^c
Level 3	Representations of \mathbb{T} -intervals, e.g. by two floating point numbers.	Representation level.
		<i>“encodes”</i> \uparrow partial, many-to-one, onto ^d
Level 4	Encodings 0111000...	Bit string level.

TABLE 1. Specification levels for interval arithmetic

4. Structure of the standard in levels

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 9, 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 this standard’s exception handling mechanism.

Level 2, in Clause 11, 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* \mathbb{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 14, 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.

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.

- Each mathematical interval (indeed each subset of \mathbb{R}) has a unique interval datum as its *\mathbb{T} -hull*—a minimal enclosing interval of the type \mathbb{T} .
- Each interval datum is a mathematical interval, if one ignores its type-tag.
- Not every interval object necessarily represents an interval datum, but when it does, that datum is unique. Each interval datum has at least one representation, and may have more than one.
- 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 shall provide at least one flavor. For brevity, phrases such as “A flavor shall provide, or document, a feature” mean that the implementation of that flavor shall provide the feature, or its documentation describe it.

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.

A flavor is identified by a unique name. Certain flavors, termed **included**, are specified in this standard. The (*list to be confirmed*) flavors are the currently included flavors. The procedure for submitting a new flavor for inclusion is described in Annex B. A conforming implementation that provides one or more included flavors may also provide non-included flavors, without losing conformance for the included flavors.

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

- (i) 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 Clause 16, Clause 17.
- (ii) There is a set of *common intervals* whose members are—in a sense made precise below—intervals of any flavor.
- (iii) 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.

The result in item (iii) is a mathematically tightest (Level 1) result, ignoring any interval widening due to finite precision (Level 2).

5.2. Definition of common intervals and common evaluations. The choice of the set of common intervals, and the set of common evaluations of an operation, is a design decision that defines the flavor concept. It should aim for simplicity, and the common evaluations 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 [5] on the set \mathbb{IR} of *closed bounded nonempty real intervals*, and no other intervals belong to all of them. Hence, the chosen set \mathcal{C} of common intervals is \mathbb{IR} .

The common evaluations are specified in terms of graphs of interval operations. For an interval operation φ 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 $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k; \mathbf{y})$ such that $\varphi(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k) = \mathbf{y}$ is true in that flavor. Each such tuple is called an *operation instance*.

The general rule is that each φ has a set $\mathcal{CE}(\varphi)$ of **common evaluations**: operation instances $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k; \mathbf{y})$ such that all its components are in \mathbb{IR} and

$$\varphi(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k) = \mathbf{y} \quad \text{shall hold in all flavors.}$$

$\mathcal{CE}(\varphi)$ may be regarded as the *flavor-independent graph* of φ . For brevity, writing

$$\text{the evaluation } \varphi(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k) = \mathbf{y} \text{ is common,} \quad (1)$$

or the equivalent notation when φ is an infix operator (e.g., $\mathbf{x}_1 + \mathbf{x}_2 = \mathbf{y}$), means that $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k; \mathbf{y}) \in \mathcal{CE}(\varphi)$.

The standard defines $\mathcal{CE}(\varphi)$ as follows.

Arithmetic operation: that is, an interval extension of the corresponding point function φ . The common operation instances are those $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k; \mathbf{y})$ such that the point function φ is *defined and continuous at each point of* the closed, bounded, nonempty box $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k)$, and \mathbf{y} equals the range of φ over this box. Then necessarily \mathbf{y} belongs to \mathbb{IR} .

Non-arithmetic operation: The common operation instances are those tuples with common inputs \mathbf{x}_i such that *the result \mathbf{y} is also common*. In particular for **convexHull**, the common operation instances are those $(\mathbf{x}_1, \mathbf{x}_2; \mathbf{y})$ with arbitrary $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{IR}$ and \mathbf{y} equal to the convex hull of $\mathbf{x}_1 \cup \mathbf{x}_2$. For **intersection**, they are those $(\mathbf{x}_1, \mathbf{x}_2; \mathbf{y})$ with

arbitrary $x_1, x_2 \in \mathbb{IR}$ and y equal to $x_1 \cap x_2$, provided the latter is *nonempty* (since $\emptyset \notin \mathbb{IR}$).

[Examples.

- The evaluation $[-1, 4]/[3, 4] = [-1/3, 4/3]$ is common; but $[3, 4]/[-1, 4] = y$ is not common, for any $y \in \mathbb{IR}$.
- In the set-based flavor, $\sqrt{[-1, 4]} = [0, 2]$. But in the Kaucher flavor $\sqrt{[-1, 4]}$ is undefined, so $\sqrt{[-1, 4]} = y$ cannot be common for any y . In fact $\mathcal{CE}(\text{sqrt})$ is the set of all $([a, b]; [\sqrt{a}, \sqrt{b}])$ for which $0 \leq a \leq b < +\infty$.
- Similarly, $([-1, 4] \cap [5, 6] = \emptyset)$ in the set-based flavor, while $([-1, 4] \cap [5, 6] = [5, 4])$ in the Kaucher flavor. Thus $([-1, 4] \cap [5, 6] = y)$ cannot be common for any y .
- The above definition for arithmetic operations requires R1 “ φ is defined and continuous at each point of x ”, which is a stronger constraint on $\mathcal{CE}(\varphi)$ (results in fewer common evaluations) than R2 “the restriction of φ to x is everywhere defined and continuous”. This produces a weaker constraint on what interval arithmetics can be flavors (with fewer rules, more arithmetics obey them). In particular, requirement R1 permits cset arithmetic to be a flavor, while R2 does not.

E.g., the common evaluations of the function $\text{floor}(x)$ are all $([a, b]; [k, k])$ with $k \in \mathbb{Z}$, $k < a \leq b < k + 1$. Thus $\text{floor}([1, 1.9]) = [1, 1]$ is not common, because $\text{floor}()$ is not continuous at 1, despite its restriction to $[1, 1.9]$ being everywhere continuous. If it were required to be common, cset arithmetic could not be a flavor.

]

5.3. Loose common evaluations. At Level 2, common evaluations are usually not computable because of roundoff; instead, an enclosing interval of some finite precision interval type is computed. The notion of a **loose common evaluation** of an operation φ takes account of this: it is defined to be any

$$\varphi(x_1, x_2, \dots, x_k) = y' \quad (2)$$

where $\varphi(x_1, x_2, \dots, x_k) = y$ is common and y' is a member of \mathbb{IR} containing y . A member of $\mathcal{CE}(\varphi)$ may be called **tight**, to emphasize that it is not loose. The set of loose common evaluations is uniquely determined by the set of (tight) common evaluations.

Informally, for a given φ and $x = (x_1, x_2, \dots, x_k)$, the loose common evaluations describe all closed bounded intervals that might be produced by evaluating an enclosure of $\text{Rge}(\varphi|x)$ in finite precision. [Note. Different qualities of Level 2 enclosure are distinguished by the terms tightest, accurate and valid introduced in §11.10.]

5.4. Relation of common evaluations to flavors. The formal definition of common evaluations takes 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 $x \in \mathbb{IR}$.

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

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* $\mathbb{IR} \rightarrow \mathfrak{F}$. Usually, $f(x)$ is abbreviated to $\mathfrak{f}x$.

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

$$\begin{aligned} &\text{If } (x_1, x_2, \dots, x_k; y) \text{ is a common operation instance of } \varphi, \\ &\text{then } (\mathfrak{f}x_1, \mathfrak{f}x_2, \dots, \mathfrak{f}x_k; \mathfrak{f}y) \text{ is an operation instance of } \varphi \text{ in flavor } \mathfrak{F}. \end{aligned} \quad (3)$$

That is, if the evaluation $\varphi(x_1, x_2, \dots, x_k) = y$ is common, then $\varphi(\mathfrak{f}x_1, \mathfrak{f}x_2, \dots, \mathfrak{f}x_k)$ must be defined in \mathfrak{F} with value $\mathfrak{f}y$.

An evaluation in \mathfrak{F} of an expression, in which only (loose) common evaluations of elementary operations occur, is called a common evaluation of that expression. That is, in a flavor (\mathfrak{F}, f) , the expression’s inputs are members of $f(\mathbb{IR})$, and each intermediate value is produced by a common evaluation of an operation so that it is also in $f(\mathbb{IR})$; hence the final result is in $f(\mathbb{IR})$.

The `com` decoration makes it possible to determine, for a specific expression and specific interval inputs, whether common evaluation has occurred, see Clause 7.

5.5. Flavors and the Fundamental Theorem. Suppose f is an arithmetic expression (does not contain set operations such as intersection) so that it defines a real point function $f(x_1, \dots, x_n)$, and is evaluated in classical interval arithmetic over a box $\mathbf{x} = (x_1, \dots, x_n)$ where the x_i are in \mathbb{IR} . The classical FTIA (§) can be stated in the following form:

If the evaluation is common then f is everywhere defined and continuous on \mathbf{x} ,
and the output \mathbf{y} encloses the range of f over \mathbf{x} .

This holds mathematically (at Level 1) using tight evaluations of individual operations in the expression, and also in finite precision (Level 2), using loose evaluations.

It also is 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 $y^* \in \mathfrak{F}$. 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 $x_j \in \mathbb{IR}$ and $y \in \mathbb{IR}$, and draw the conclusions of the classical FTIA.

[Note. Besides this “minimal” FTIA for any flavor, which derives automatically from the classical FTIA and the meaning of common evaluation, each of the set-based and Kaucher flavors has a more general FTIA that contains the minimal FTIA as a special case. Note the Kaucher flavor has a generalized meaning of the “contains” relation, which is used in stating its FTIA: $[a, b] \supseteq [c, d]$ means $(a \leq c \wedge b \geq d)$, whether $[a, b]$ and $[c, d]$ are proper or improper. Since the minimal FTIA, above, is defined by mapping back to common intervals, it does not need to use such generalized notions.]

It is useful to consider when a *flavor-independent result* of common evaluation occurs. That is, when does evaluating a given expression, over the “same” bounded nonempty box in two different flavors, give identical results modulo the embedding map? At Level 1, this is always true if the individual operation evaluations are tight. Also, an implementation may create conditions under which it holds in finite precision, by the flavors “sharing” at Level 2 as follows:

- It provides some shared interval types, which represent exactly the same finite set of common intervals in each flavor.
- It provides some shared library operations (on the shared types), which when acting on common intervals have identical rounding behavior in each flavor, modulo the embedding map.

Then a common evaluation that only uses shared types and operations gives identical results in both flavors, modulo the embedding map.

6. Expressions and the functions they define

An expression is some symbolic form used to define a function. Expressions are central to interval computation, because the Fundamental Theorem of Interval Arithmetic (FTIA) is about interpreting an expression in different ways: first, as defining a mathematical real point function f ; second, as defining interval mappings that give proven enclosures for the range of f over an input box \mathbf{x} . (The meaning of “enclosure” is the same, independent of flavor, for common intervals, but is flavor-defined in general.) If decorated intervals are used, applying the Fundamental Theorem of Decorated Interval Arithmetic (FTDIA) also gives information about definedness, continuity, etc., of f on \mathbf{x} . The term FT[D]IA is used below to mean both of these theorems.

The standard specifies behavior at the individual operation level, that makes it possible to draw the conclusions of the FT[D]IA. It does not define a meaning of “expression” since this would raise language-dependent semantic issues that are outside its scope.

However it is useful to list some things that an expression is and is not, to give a context for the specifications of the standard:

- (i) An expression f defines a relation between certain *inputs*, often represented by mathematical or programming variables, and certain *outputs*, via the application of named *operations* that come from a *library*. f may be represented in various ways, e.g., by a normal algebraic formula or by a segment of program code or pseudo-code or as a computational graph. This is illustrated for (an expression defining) the function $f(x, y) = \sqrt{y^2 - 1} + xy$ in Figure 1 (a,b,c) respectively.

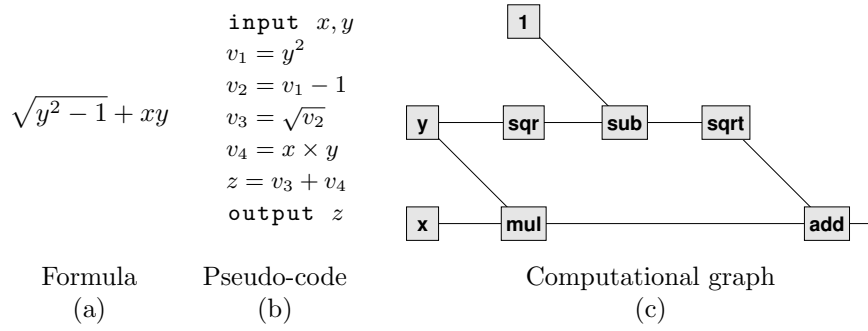


FIGURE 1. Essentially equivalent notations for an expression. The pseudo-code (b) breaks the formula into single library operations, and uses static single-assignment (SSA) form. The graph form (c) uses the names in §9.6.3, for library operations.

- Code may define a *vector* expression, with several outputs; however, all the individual library operations of the standard are *scalar*, with possibly several inputs but a single output.
- (ii) For the FT[D]IA to apply, the operations must be *generic*. The expression is an *arithmetic expression*, each of whose operations is primarily a *point-operation* with real-number input and output, but which has at Level 1 an *interval version* with interval input and output, and a corresponding *decorated interval version*. Using these different versions gives *point evaluation*, *interval evaluation* and *decorated interval evaluation* of the expression, also termed evaluation in *point mode*, *interval mode* or *decorated interval mode*. At Level 2, each such version typically splits into further versions corresponding to different finite-precision interval types provided by the implementation.

The set operations **intersection** and **convexHull** are not point-operations and cannot appear in an arithmetic expression. However they are useful for efficiently *implementing* interval versions of functions defined piecewise (see Example ii in §10.9).

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 §9.6, §9.7 and in Clause 11.

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

- (iii) The expression must be *explicit*. Interval analysis has ways to find range-enclosures for a function such as $y = g(x)$ defined implicitly by solving $\sqrt{y^2 - 1} + xy = 0$ for y ; but the FT[D]IA does not apply directly to such functions.
- (iv) The computational graph of the expression is a directed acyclic graph. When program code is involved, the FT[D]IA applies to the expression f evaluated in a particular execution of that code. If code contains conditional statements or loops, f and its graph may change from one execution to another. Often such code is designed to define a function piecewise (e.g., over several intervals, see example in §10.9). The user is responsible for checking that the FT[D]IA applies as intended in such cases; the standard provides no way to check automatically that a property such as continuity holds globally.
- (v) There is a mismatch between expressions and set theory when doing interval-evaluation, illustrated by the following. Define a 3-argument function by

$$f(x, y, z) = x + z, \quad (4)$$

where argument y does not occur in the expression. Interval-evaluation $w = f(x, y, z)$ returns $x + z$, ignoring y . When y is empty, e.g.

$$w = f([1, 2], \emptyset, [3, 4]) = [4, 6], \quad (5)$$

this is at odds with set-theory, which says w encloses the range of the point-function over the product box $[1, 2] \times \emptyset \times [3, 4]$. Since this is empty, the range is empty, suggesting the computed w should be \emptyset instead of $[4, 6]$.

It is language-defined which of these interpretations is adopted, but the FT[D]IA theory in Annex C is framed so that what interval evaluation actually computes—here [4, 6]—is correct. This is done by defining the function specified by an expression to depend only on those inputs that in the evaluation’s computational graph lie on a path to the output. In (4), the function thus has two arguments x, z instead of three. This is conveniently handled by using *keyword association* of dummy to actual arguments—for expressions, not for individual library operations for which normal positional arguments are used—so that irrelevant variables can be skipped. E.g. the argument association in (5) may be written $f(x = [1, 2], z = [3, 4])$. See Annex C for details.

7. Decoration system

7.1. Decorations overview. A decoration is information attached to an interval; the combination is called a decorated interval. Interval calculation has two main objectives:

- obtaining correct range enclosures for a real-valued function of real variables;
- verifying the assumptions of existence, uniqueness, or nonexistence theorems.

Traditional interval analysis targets the first objective; decorated intervals, as defined in this standard, target the second.

A decoration primarily describes a property, not of the interval it is attached to, but of the function defined by some code that produced the interval by evaluating over some input box.

For instance, if a section of code defines the expression $\sqrt{y^2 - 1} + xy$, then decorated-interval evaluation of this code with suitably initialized input intervals \mathbf{x}, \mathbf{y} gives information about the definedness, continuity, etc. of the point function $f(x, y) = \sqrt{y^2 - 1} + xy$ over the box (\mathbf{x}, \mathbf{y}) in the plane.

The decoration system is designed in a way that naive users of interval arithmetic do not notice anything about decorations, unless they inquire explicitly about their values. They only need

- call the `newDec` operation on the inputs of any function evaluation used to invoke an existence theorem,
- explicitly convert relevant floating-point constants (but not integer parameters such as the p in `pow \mathbf{n} (x, p) = x^p`) to intervals,

and have the full rigor of interval calculations available. A smart implementation may even relieve users from these tasks. Expert users can inspect, set and modify decorations to improve code efficiency, but are responsible for checking that computations done in this way remain rigorously valid.

Especially in the set-based flavor, decorations are based on the desire that, from an interval evaluation of a real function f on a box \mathbf{x} , one should get not only a range enclosure $f(\mathbf{x})$ but also a guarantee that the pair (f, \mathbf{x}) has certain important properties, such as $f(x)$ being defined for all $x \in \mathbf{x}$, f restricted to \mathbf{x} being continuous, etc. This goal is achieved, in parts of a program that require it, by performing *decorated interval evaluation*, whose semantics is summarized as follows:

Each intermediate step of the original computation depends on some or all of the inputs, so it can be viewed as an intermediate function of these inputs. The result interval obtained on each intermediate step is an enclosure for the range of the corresponding intermediate function. The decoration attached to this intermediate interval reflects the available knowledge about whether this intermediate function is guaranteed to be everywhere defined, continuous, bounded, etc., on the given inputs.

In some flavors, certain interval operations ignore decorations, i.e., give undecorated interval output. Users are responsible for the appropriate propagation of decorations by these operations.

The function f is assumed to be expressed by code, an algebraic formula, etc.—generically termed an *expression*—which can be evaluated in several modes: point evaluation, interval evaluation, or decorated interval evaluation. The standard does not specify a definition of “expression”; however, Annex C gives formal proofs in terms of a particular definition, and indicates how this relates to expressions in some programming languages.

This standard’s decoration model, in contrast with 754’s, has no status flags. A general aim, as in 754’s use of NaN and flags, is not to interrupt the flow of computation: rather, to collate information while evaluating f , that can be inspected afterwards. This enables a fully local

handling of exceptional conditions in interval calculations—important in a concurrent computing environment.

An implementation may provide any of the following: (i) status flags that are raised in the event of certain decoration values being produced by an operation; (ii) means for the user to specify that such an event signals an exception, and to invoke a system- or user-defined handler as a result. [Example. The user may be able to specify execution be terminated if an arithmetic operation is evaluated on a box that is not wholly inside its domain—an interval version of 754’s “invalid operation” exception.] Such features are language- or implementation-defined.

7.2. Decoration definition and propagation. Each flavor shall document its set of provided decorations and their mathematical definitions. These are flavor-defined, with the exception of the decoration `com`, see §7.3.

The implementation makes the decoration system of each flavor available to the user via *decorated interval extensions* of relevant library operations. Such an operation φ , with interval inputs $\mathbf{x}_1, \dots, \mathbf{x}_k$ carrying decorations dx_1, \dots, dx_k , shall compute the same interval output \mathbf{y} as the corresponding bare interval extension of φ —hence dependent on the \mathbf{x}_i but not on the dx_i . It shall compute a *local decoration* d , dependent on the \mathbf{x}_i and possibly on \mathbf{y} , but not on the dx_i . It shall combine d with the dx_i by a flavor-defined *propagation rule* to give an output decoration dy , and return \mathbf{y} decorated by dy .

The local decoration d may convey purely Level 1 information—e.g., that φ is everywhere continuous on the box $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_k)$. It may convey Level 2 information related to the particular finite-precision interval types being used—e.g., that \mathbf{y} , though mathematically a bounded interval, became unbounded by overflow. For diagnostic use it may convey Level 3 or 4 information, e.g., how an interval is represented, or how memory is used.

If f is an expression, decorated interval evaluation of an expression means evaluation of f with decorated interval inputs and using decorated interval extensions of the expression’s library operations. Those inputs generally need to be given suitable initial decorations that lead to the most informative output-decoration. A flavor shall provide a `newDec` function for this purpose. If \mathbf{x} is a bare interval, `newDec(\mathbf{x})` equals \mathbf{x} with such an initial decoration. If \mathbf{x} is a decorated interval, the decoration is discarded and `newDec` applied to the bare interval part.

It is the responsibility of each flavor to document the meaning of its decorations, and the correct use of these decorations within programs.

7.3. Recognizing common evaluation. A flavor may provide the decoration `com` with the following propagation rule for library arithmetic operations. In an implementation with more than one flavor, each flavor shall do so.

In the following, φ denotes an arbitrary interval extension of a point library arithmetic operation φ , provided by the implementation at Level 2 (typically the one associated with a particular interval type).

Let φ applied to input intervals $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k$ give the computed result \mathbf{y} , and let $\varphi(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k) = \mathbf{y}$ be a loose common evaluation as defined in (2). If each of the inputs \mathbf{x}_i is decorated `com`, then the output \mathbf{y} shall be decorated `com`.

Informally, `com` records that the individual operation φ took bounded nonempty input intervals and produced a bounded (necessarily nonempty) output interval. This can be interpreted as indicating “overflow did not occur”. Further, the propagation rule ensures that if the initial inputs to an arithmetic expression f are bounded and nonempty, and are initialized with the decoration `com`, then the final result $\mathbf{y} = f(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k)$ is decorated `com` if and only if the evaluation of the whole expression was common as defined in §5.4.

Flavors should define other decoration values, but `com` is the only one that is required to have the same meaning in all flavors.

[Examples. Reasons why an individual evaluation of φ with common inputs $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_k)$ may not return `com` include the following.

Outside domain: The implementation finds φ is not defined and continuous everywhere on \mathbf{x} .

Examples: $\sqrt{[-4, 4]}$, $\text{sign}([0, 2])$.

Overflow: *The Level 1 result is too large to be represented. Example: Consider an interval type \mathbb{T} whose intervals are represented by their lower and upper bounds in some floating point format, let REALMAX be the largest finite number in that format, and x be the common \mathbb{T} -interval $[0, \text{REALMAX}]$. Then $x + x$ cannot be enclosed in a common \mathbb{T} -interval.*

Cost: *It is too expensive to determine whether the result is too large to be represented. A possible example is $\tan([a, b])$ where $[a, b]$ is of an interval type \mathbb{T} as in the previous item, and one of its endpoints happens to be very close to a singularity of $\tan(x)$.*

]

8. Conformance requirements

To be completed later.

DRAFT 7.0

DRAFT 7.0

CHAPTER 2

Set-Based Intervals

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

9. Level 1 description

In this clause, subclauses §9.1 to §9.5 describe the theory of mathematical intervals and interval functions that underlies this flavor. 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 §9.6, 9.7 list the required and recommended *arithmetic operations* (also called elementary functions) with their mathematical specifications. Clause 10 describes, at a mathematical level, the system of *decorations* that is used among other things for exception handling in this flavor of the standard.

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

- The set $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$ of **extended reals**. Following the terminology of 754 (e.g., 754§2.1.25), any member of $\overline{\mathbb{R}}$ 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, there are 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.

9.2. Intervals. The set of mathematical intervals supported by this flavor is denoted $\overline{\mathbb{IR}}$. It consists of exactly those subsets of the real line \mathbb{R} that are closed and connected in the topological sense. Thus it comprises the empty set (denoted \emptyset or Empty) together with all nonempty closed intervals x of real numbers having the form

$$x = [\underline{x}, \overline{x}] := \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\}, \quad (6)$$

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

[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 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 §9.6, 9.7.
- The set of intervals $\overline{\mathbb{IR}}$ 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 $\underline{x} > \overline{x}$, and also as $[-\infty, -\infty]$ or $[+\infty, +\infty]$. The description (6) 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 $\overline{\mathbb{I}}$ is an operator: for any subset S of the extended reals $\overline{\mathbb{R}}$, the set $\overline{\mathbb{I}}S$ or $\overline{\mathbb{I}}(S)$ comprises Empty together with all intervals (6) whose bounds belong to S . This is used at Level 2 to specify finite-precision types, e.g. $\overline{\mathbb{I}}(\text{binary64})$ which comprises Empty together with all nonempty intervals whose bounds are IEEE754 double precision numbers.

- The smaller set that excludes Empty, Entire, and semi-bounded intervals is indicated by the notation \mathbb{I} , e.g. \mathbb{IR} or $\mathbb{I}(\mathbb{R})$, the set of nonempty closed bounded intervals, is the set used in classical interval literature such as the work of R.E. Moore.

]

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

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

9.4. Functions.

9.4.1. *Function terminology.* In this flavor, operations are written as named functions; in a specific implementation they might be represented by operators (e.g., using an infix notation), or by families of type-specific functions, or by operators or functions whose names might differ from those used here.

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

- A *point function* (§9.4.2) is a mathematical real function of real variables. Otherwise, *function* is usually used with its general mathematical meaning.
- A (point) *arithmetic operation* (§9.4.2) is a mathematical real function for which an implementation provides versions in the implementation's *library* (§9.4.2).
- A *version* of a point function f means a function derived from f ; typically a bare or decorated interval extension (§9.4.3) of f .
- An *interval arithmetic operation* is an interval extension of a point arithmetic operation (§9.4.3).
- An *interval non-arithmetic operation* is an interval-to-interval library function that is not an interval arithmetic operation (§9.4.3).
- A *constructor* is a function that creates an interval from non-interval data (§9.6.8).

9.4.2. *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 $\text{Dom } f$. To specify n , call f an n -variable point function, or denote values of f as

$$f(x_1, \dots, x_n). \quad (7)$$

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

$$\text{Rge}(f | \mathbf{s}) := \{ f(x) \mid x \in \mathbf{s} \text{ and } x \in \text{Dom } f \}. \quad (8)$$

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

Equivalently, for the case where f takes separate arguments $\mathbf{s}_1, \dots, \mathbf{s}_n$, each being a subset of \mathbb{R} , the range is written as $\text{Rge}(f \mid \mathbf{s}_1, \dots, \mathbf{s}_n)$. This is an alternative notation when \mathbf{s} is the cartesian product of the \mathbf{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.

9.4.3. *Interval-valued functions.* A box is an interval vector $\mathbf{x} = (x_1, \dots, x_n) \in \mathbb{IR}^n$. It is usually identified with the cartesian product $x_1 \times \dots \times x_n \subseteq \mathbb{R}^n$; 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 \mathbf{f} from n -dimensional boxes to intervals, that is $\mathbf{f} : \mathbb{IR}^n \rightarrow \mathbb{IR}$, such that $f(x) \in \mathbf{f}(\mathbf{x})$ whenever $x \in \mathbf{x}$ and $f(x)$ is defined, equivalently

$$\mathbf{f}(\mathbf{x}) \supseteq \text{Rge}(f | \mathbf{x})$$

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

$$\mathbf{f}(\mathbf{x}) := \text{hull}(\text{Rge}(f \mid \mathbf{x})).$$

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

$$\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \supseteq \text{Rge}(f \mid \mathbf{x}_1, \dots, \mathbf{x}_n),$$

and the natural interval extension is defined by

$$\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) := \text{hull}(\text{Rge}(f \mid \mathbf{x}_1, \dots, \mathbf{x}_n))$$

for any intervals $\mathbf{x}_1, \dots, \mathbf{x}_n$.

In some contexts it is useful for \mathbf{x} to be a general subset of \mathbb{R}^n , or the \mathbf{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, Clause 10, gives a 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

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

[*Example. With these definitions, the relevant natural interval extensions satisfy $\sqrt{[-1, 4]} = [0, 2]$ and $\sqrt{[-2, -1]} = \emptyset$; also $\mathbf{x} \times [0, 0] = [0, 0]$ for any nonempty \mathbf{x} , and $\mathbf{x}/[0, 0] = \emptyset$, for any \mathbf{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, $(\mathbf{x}, \mathbf{y}) \mapsto \mathbf{x} \cap \mathbf{y}$ and $(\mathbf{x}, \mathbf{y}) \mapsto \text{hull}(\mathbf{x} \cup \mathbf{y})$.

9.4.4. *Constants.* A real scalar function with no arguments—a mapping $\mathbb{R}^n \rightarrow \mathbb{R}^m$ 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., π) 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 $[c, c]$.

9.5. Expressions. This flavor gives the term “expression” the general meaning described in Clause 6.

9.6. 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 §9.6.1, 9.6.3, 9.6.4, 9.6.5, each such 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 §9.6.9, is detailed in §11.9, 11.11.

The names of operations, as well as symbols used for operations (e.g., for the comparisons in §9.6.10), may not correspond to those that any particular language would use.

9.6.1. 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 §11.11.1. [Example. This includes the *inf-sup form* `[1.234e5,Inf]`; the *mid-rad form* `<3.1416+-0.00001>`; or the *named interval constant* `Entire`.]

An invalid denotation has no value at Level 1.

9.6.2. Interval constants.

The constant functions `bareempty()` and `bareentire()` return `Empty` and `Entire` respectively.

9.6.3. Forward-mode elementary functions.

Table 1 on page 19 lists required arithmetic operations. The term *operation* includes functions normally written in function notation $f(x, y, \dots)$, as well as those normally written in unary or binary operator notation, $\bullet x$ or $x \bullet y$.

[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 extensions. 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!

Notes to Table 1

- In describing the domain, notation such as $\{y = 0\}$ is short for $\{(x, y) \in \mathbb{R}^2 \mid y = 0\}$, etc.
- Regarded as a family of functions parameterized by the integer argument p .
- Defined as $e^{y \ln x}$ for real $x > 0$ and all real y , and 0 for $x = 0$ and $y > 0$, else there is no value at Level 1.
- $b = e, 2$ or 10 , respectively.
- 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.
- `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 $\langle \rangle$.
- `sign(x)` is -1 if $x < 0$; 0 if $x = 0$; and 1 if $x > 0$.
- `ceil(x)` is the smallest integer $\geq x$. `floor(x)` is the largest integer $\leq x$. `roundTiesToEven(x)`, `roundTiesToAway(x)` are the nearest integer to x , with ties rounded to the even integer or away from zero respectively. `trunc(x)` is the nearest integer to x in the direction of zero. (As defined in the C standard §7.12.9.)
- Smallest, or largest, of its real arguments. A family of functions parameterized by the arity k .

9.6.4. 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 `case(c, g, h)` is provided. To simplify defining its interval extension, the argument c specifying the condition is

TABLE 1. Required forward elementary functions.

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

Name	Definition	Point function domain	Point function range	Note
neg (x)	$-x$	\mathbb{R}	\mathbb{R}	
add (x, y)	$x + y$	\mathbb{R}^2	\mathbb{R}	
sub (x, y)	$x - y$	\mathbb{R}^2	\mathbb{R}	
mul (x, y)	xy	\mathbb{R}^2	\mathbb{R}	
div (x, y)	x/y	$\mathbb{R}^2 \setminus \{y = 0\}$	\mathbb{R}	a
recip (x)	$1/x$	$\mathbb{R} \setminus \{0\}$	$\mathbb{R} \setminus \{0\}$	
sqrt (x)	\sqrt{x}	$[0, \infty)$	$[0, \infty)$	
fma (x, y, z)	$(x \times y) + z$	\mathbb{R}^3	\mathbb{R}	
case (c, g, h)		See §9.6.4.		
sqr (x)	x^2	\mathbb{R}	$[0, \infty)$	
pown (x, p)	$x^p, p \in \mathbb{Z}$	$\begin{cases} \mathbb{R} & \text{if } p \geq 0 \\ \mathbb{R} \setminus \{0\} & \text{if } p < 0 \end{cases}$	$\begin{cases} \mathbb{R} & \text{if } p > 0 \text{ odd} \\ [0, \infty) & \text{if } p > 0 \text{ even} \\ \{1\} & \text{if } p = 0 \\ \mathbb{R} \setminus \{0\} & \text{if } p < 0 \text{ odd} \\ (0, \infty) & \text{if } p < 0 \text{ even} \end{cases}$	b
pow (x, y)	x^y	$\{x > 0\} \cup \{x = 0, y > 0\}$	$[0, \infty)$	a, c
exp, exp2, exp10 (x)	b^x	\mathbb{R}	$(0, \infty)$	d
log, log2, log10 (x)	$\log_b x$	$(0, \infty)$	\mathbb{R}	d
sin (x)		\mathbb{R}	$[-1, 1]$	
cos (x)		\mathbb{R}	$[-1, 1]$	
tan (x)		$\mathbb{R} \setminus \{(k + \frac{1}{2})\pi \mid k \in \mathbb{Z}\}$	\mathbb{R}	
asin (x)		$[-1, 1]$	$[-\pi/2, \pi/2]$	e
acos (x)		$[-1, 1]$	$[0, \pi]$	e
atan (x)		\mathbb{R}	$(-\pi/2, \pi/2)$	e
atan2 (y, x)		$\mathbb{R}^2 \setminus \{(0, 0)\}$	$(-\pi, \pi]$	e, f, g
sinh (x)		\mathbb{R}	\mathbb{R}	
cosh (x)		\mathbb{R}	$[1, \infty)$	
tanh (x)		\mathbb{R}	$(-1, 1)$	
asinh (x)		\mathbb{R}	\mathbb{R}	
acosh (x)		$[1, \infty)$	$[0, \infty)$	
atanh (x)		$(-1, 1)$	\mathbb{R}	
sign (x)		\mathbb{R}	$\{-1, 0, 1\}$	h
ceil (x)		\mathbb{R}	\mathbb{Z}	i
floor (x)		\mathbb{R}	\mathbb{Z}	i
trunc (x)		\mathbb{R}	\mathbb{Z}	i
roundTiesToEven (x)		\mathbb{R}	\mathbb{Z}	i
roundTiesToAway (x)		\mathbb{R}	\mathbb{Z}	i
abs (x)	$ x $	\mathbb{R}	$[0, \infty)$	
min (x_1, \dots, x_k)		\mathbb{R}^k for $k = 2, 3, \dots$	\mathbb{R}	j
max (x_1, \dots, x_k)		\mathbb{R}^k for $k = 2, 3, \dots$	\mathbb{R}	j

real (instead of boolean), and the condition means $c < 0$ by definition. That is,

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

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

$$\text{case}(\mathbf{c}, \mathbf{g}, \mathbf{h}) = \begin{cases} \text{if } \mathbf{c} \text{ is empty} & \text{then } \emptyset \\ \text{elseif } \mathbf{c} \text{ is a subset of the half-line } x < 0 & \text{then } \mathbf{g} \\ \text{elseif } \mathbf{c} \text{ is a subset of the half-line } x \geq 0 & \text{then } \mathbf{h} \\ \text{else} & \text{convexHull}(\mathbf{g}, \mathbf{h}). \end{cases} \quad (9)$$

for any intervals $\mathbf{c}, \mathbf{g}, \mathbf{h}$.

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

$$\mathbf{f}(x) = \text{case}(c(x), g(x), h(x)) \quad (10)$$

is automatically an interval extension of f .

[Notes.

1. Equation (9) 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 (10), it allows “short-circuiting”. That is, one can suppress evaluation of $h(x)$ if $\bar{c} < 0$, and of $g(x)$ if $\underline{c} \geq 0$. 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 (10), applied to a nonempty $x = [\underline{x}, \bar{x}]$, gives the exact range $\{|x| \mid x \in x\}$ when $\bar{x} < 0$ or $0 \leq \underline{x}$, but the poor enclosure $(-x) \cup x$ when $\underline{x} < 0 \leq \bar{x}$.

3. $\text{case}(c, g, h)$ is equivalent to the C expression $(c < 0 ? g : h)$.
4. Compound conditions may be expressed using the max and 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 $(x < 0 \text{ or } y < 0)$.

]

9.6.5. 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 φ , a **reverse interval extension** of φ is a binary interval function φRev such that

$$\varphi\text{Rev}(\mathbf{c}, \mathbf{x}) \supseteq \{x \in \mathbf{x} \mid \varphi(x) \text{ is defined and in } \mathbf{c}\}, \quad (11)$$

for any intervals \mathbf{c}, \mathbf{x} .

Similarly, a binary arithmetic operation \bullet has two forms of reverse interval extension, which are ternary interval functions $\bullet\text{Rev}_1$ and $\bullet\text{Rev}_2$ such that

$$\bullet\text{Rev}_1(\mathbf{b}, \mathbf{c}, \mathbf{x}) \supseteq \{x \in \mathbf{x} \mid b \in \mathbf{b} \text{ exists such that } x \bullet b \text{ is defined and in } \mathbf{c}\}, \quad (12)$$

$$\bullet\text{Rev}_2(\mathbf{a}, \mathbf{c}, \mathbf{x}) \supseteq \{x \in \mathbf{x} \mid a \in \mathbf{a} \text{ exists such that } a \bullet x \text{ is defined and in } \mathbf{c}\}. \quad (13)$$

If \bullet is commutative then $\bullet\text{Rev}_1$ and $\bullet\text{Rev}_2$ agree and may be implemented simply as $\bullet\text{Rev}$.

In each of (11, 12, 13), the unique **natural reverse interval extension** is the one whose value is the interval hull of the right-hand side. Clearly, any reverse interval extension encloses this hull.

The last argument \mathbf{x} in each of (11, 12, 13) is optional, with default $\mathbf{x} = \mathbb{R}$ if absent.

[Note. The argument \mathbf{x} can be thought of as giving prior knowledge about the range of values taken by a point-variable x , which is then sharpened by applying the reverse function: see the example below.]

Reverse operations shall be provided as in Table 2. Note $\text{pownRev}(x, p)$ is regarded as a family of unary functions parametrized by p .

[Example.

TABLE 2. Required reverse elementary functions.

From unary functions	From binary functions
$\text{sqrRev}(c, x)$	$\text{mulRev}(b, c, x)$
$\text{recipRev}(c, x)$	$\text{divRev1}(b, c, x)$
$\text{absRev}(c, x)$	$\text{divRev2}(a, c, x)$
$\text{pownRev}(c, x, p)$	$\text{powRev1}(b, c, x)$
$\text{sinRev}(c, x)$	$\text{powRev2}(a, c, x)$
$\text{cosRev}(c, x)$	$\text{atan2Rev1}(b, c, x)$
$\text{tanRev}(c, x)$	$\text{atan2Rev2}(a, c, x)$
$\text{coshRev}(c, x)$	

- Consider the function $\text{sqr}(x) = x^2$. Evaluating $\text{sqrRev}([1, 4])$ answers the question: given that $1 \leq x^2 \leq 4$, what interval can we restrict x to? Using the natural reverse extension, we have

$$\text{sqrRev}([1, 4]) = \text{hull}\{x \in \mathbb{R} \mid x^2 \in [1, 4]\} = \text{hull}([-2, -1] \cup [1, 2]) = [-2, 2].$$

- If we can add the prior knowledge that $x \in x = [0, 1.2]$, then using the optional second argument gives the tighter enclosure

$$\text{sqrRev}([1, 4], [0, 1.2]) = \text{hull}\{x \in [0, 1.2] \mid x^2 \in [1, 4]\} = \text{hull}([0, 1.2] \cap ([-2, -1] \cup [1, 2])) = [1, 1.2].$$

- 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 above, this method evaluates

$$\text{sqrRev}([1, 4]) \cap x = [-2, 2] \cap [0, 1.2] = [0, 1.2],$$

so no tightening of the enclosure x is obtained.

]

9.6.6. Cancellative addition and subtraction.

⚠ 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 $y + z$.

[Example. In some applications one has a list of intervals a_1, \dots, a_n , and needs to form each interval s_k which is the sum of all the a_i except a_k , that is $s_k = \sum_{i=1, i \neq k}^n a_i$, for $k = 1, \dots, n$.

Evaluating all these sums independently costs $O(n^2)$ work. However, if one forms the sum s of all the a_i , one can obtain each s_k from s and a_k by cancellative subtraction. This method only costs $O(n)$ work.

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 $\text{cancelPlus}(x, y)$, equivalent to $\text{cancelMinus}(x, -y)$ and therefore not specified separately.

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

$$y + z \supseteq x \tag{14}$$

if such a z exists. Otherwise $\text{cancelMinus}(x, y)$ 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 $x = [\underline{x}, \bar{x}]$ and $y = [\underline{y}, \bar{y}]$ are both nonempty and bounded, define

$\underline{z} = \underline{x} - \underline{y}$ and $\bar{z} = \bar{x} - \bar{y}$. Then \mathbf{z} is defined to be $[\underline{z}, \bar{z}]$ if $\underline{z} \leq \bar{z}$ (equivalently if $\text{width}(\mathbf{x}) \geq \text{width}(\mathbf{y})$), and has no value otherwise. If either \mathbf{x} or \mathbf{y} is unbounded, \mathbf{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 §14.5.]

9.6.7. Non-arithmetic (set) operations.

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

Name	Definition
<code>intersection(x, y)</code>	$\mathbf{x} \cap \mathbf{y}$
<code>convexHull(x, y)</code>	interval hull of $\mathbf{x} \cup \mathbf{y}$

9.6.8. 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 §9.2) 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 13 on I/O and may optionally be followed.]

9.6.9. 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 `mid(x)` and `rad(x)` simultaneously.]

TABLE 3. Required numeric functions of an interval $\mathbf{x} = [\underline{x}, \bar{x}]$.

Note `inf` can return $-\infty$; each of `sup`, `wid`, `rad` and `mag` can return $+\infty$.

Name	Definition
<code>inf(x)</code>	$\begin{cases} \text{lower bound of } \mathbf{x} \text{ if } \mathbf{x} \text{ is nonempty} \\ \infty \text{ if } \mathbf{x} \text{ is empty} \end{cases}$
<code>sup(x)</code>	$\begin{cases} \text{upper bound of } \mathbf{x} \text{ if } \mathbf{x} \text{ is nonempty} \\ -\infty \text{ if } \mathbf{x} \text{ is empty} \end{cases}$
<code>mid(x)</code>	$\begin{cases} \text{midpoint } (\underline{x} + \bar{x})/2 \text{ if } \mathbf{x} \text{ is nonempty bounded} \\ \text{no value if } \mathbf{x} \text{ is empty or unbounded} \end{cases}$
<code>wid(x)</code>	$\begin{cases} \text{width } \bar{x} - \underline{x} \text{ if } \mathbf{x} \text{ is nonempty} \\ \text{no value if } \mathbf{x} \text{ is empty} \end{cases}$
<code>rad(x)</code>	$\begin{cases} \text{radius } (\bar{x} - \underline{x})/2 \text{ if } \mathbf{x} \text{ is nonempty} \\ \text{no value if } \mathbf{x} \text{ is empty} \end{cases}$
<code>mag(x)</code>	$\begin{cases} \text{magnitude } \sup\{ x \mid x \in \mathbf{x}\} \text{ if } \mathbf{x} \text{ is nonempty} \\ \text{no value if } \mathbf{x} \text{ is empty} \end{cases}$
<code>mig(x)</code>	$\begin{cases} \text{mignitude } \inf\{ x \mid x \in \mathbf{x}\} \text{ if } \mathbf{x} \text{ is nonempty} \\ \text{no value if } \mathbf{x} \text{ is empty} \end{cases}$

△ 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.

9.6.10. Boolean functions of intervals.

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

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

TABLE 4. Comparisons for intervals \mathbf{a} and \mathbf{b} . Notation \forall_a means “for all a in \mathbf{a} ”, and so on. In column 4, $\mathbf{a}=[\underline{a}, \bar{a}]$ and $\mathbf{b}=[\underline{b}, \bar{b}]$, where $\underline{a}, \underline{b}$ may be $-\infty$ and \bar{a}, \bar{b} may be $+\infty$; and $<'$ is the same as $<$ except that $-\infty <' -\infty$ and $+\infty <' +\infty$ are true.

Name	Symbol	Definition	For $\mathbf{a}, \mathbf{b} \neq \emptyset$	Description
<code>isEqual(\mathbf{a}, \mathbf{b})</code>	$\mathbf{a} = \mathbf{b}$	$\forall_a \exists_b a = b \wedge \forall_b \exists_a b = a$	$\underline{a} = \underline{b} \wedge \bar{a} = \bar{b}$	\mathbf{a} equals \mathbf{b}
<code>containedIn(\mathbf{a}, \mathbf{b})</code>	$\mathbf{a} \subseteq \mathbf{b}$	$\forall_a \exists_b a = b$	$\underline{b} \leq \underline{a} \wedge \bar{a} \leq \bar{b}$	\mathbf{a} is a subset of \mathbf{b}
<code>less(\mathbf{a}, \mathbf{b})</code>	$\mathbf{a} \leq \mathbf{b}$	$\forall_a \exists_b a \leq b \wedge \forall_b \exists_a a \leq b$	$\underline{a} \leq \underline{b} \wedge \bar{a} \leq \bar{b}$	\mathbf{a} is weakly less than \mathbf{b}
<code>precedes(\mathbf{a}, \mathbf{b})</code>	$\mathbf{a} < \mathbf{b}$	$\forall_a \forall_b a < b$	$\bar{a} \leq \underline{b}$	\mathbf{a} is to left of but may touch \mathbf{b}
<code>isInterior(\mathbf{a}, \mathbf{b})</code>	$\mathbf{a} @ \mathbf{b}$	$\forall_a \exists_b a < b \wedge \forall_a \exists_b b < a$	$\underline{b} <' \underline{a} \wedge \bar{a} <' \bar{b}$	\mathbf{a} is interior to \mathbf{b}
<code>strictlyLess(\mathbf{a}, \mathbf{b})</code>	$\mathbf{a} < \mathbf{b}$	$\forall_a \exists_b a < b \wedge \forall_b \exists_a a < b$	$\underline{a} <' \underline{b} \wedge \bar{a} <' \bar{b}$	\mathbf{a} is strictly less than \mathbf{b}
<code>strictlyPrecedes(\mathbf{a}, \mathbf{b})</code>	$\mathbf{a} < \mathbf{b}$	$\forall_a \forall_b a < b$	$\bar{a} < \underline{b}$	\mathbf{a} is strictly to left of \mathbf{b}
<code>areDisjoint(\mathbf{a}, \mathbf{b})</code>	$\mathbf{a} \nmid \mathbf{b}$	$\forall_a \forall_b a \neq b$	$\bar{a} < \underline{b} \vee \bar{b} < \underline{a}$	\mathbf{a} and \mathbf{b} are disjoint

The following table shows what the definitions imply when at least one interval is empty.

	$\mathbf{a} = \emptyset$ $\mathbf{b} \neq \emptyset$	$\mathbf{a} \neq \emptyset$ $\mathbf{b} = \emptyset$	$\mathbf{a} = \emptyset$ $\mathbf{b} = \emptyset$
$\mathbf{a} = \mathbf{b}$	0	0	1
$\mathbf{a} \subseteq \mathbf{b}$	1	0	1
$\mathbf{a} \leq \mathbf{b}$	0	0	1
$\mathbf{a} < \mathbf{b}$	1	1	1
$\mathbf{a} @ \mathbf{b}$	1	0	1
$\mathbf{a} < \mathbf{b}$	0	0	1
$\mathbf{a} < \mathbf{b}$	1	1	1
$\mathbf{a} \nmid \mathbf{b}$	1	1	1

[Note.

- All these relations, except $\mathbf{a} \nmid \mathbf{b}$, are transitive for nonempty intervals.
- The first three are reflexive.
- `isInterior` uses the topological definition: \mathbf{b} is a neighbourhood of each point of \mathbf{a} . This implies, for instance, that `isInterior(Entire,Entire)` is true.
- In fact all occurrences of $<$ in column 4 of Table 4 can be replaced by $<'$.]

9.6.11. Dot product function.

For point vectors $x = (x_1, \dots, x_n)$ and $y = (y_1, \dots, y_n)$ the function `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 11.11.11 which deals with the Complete Arithmetic datatype and operations.

9.7. Recommended operations.

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.

9.7.1. 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 $\{x \in \mathbb{R} \mid -1 < x \leq 1\}$.

Name	Definition	Point function domain	Point function range	Note
$\text{rootn}(x, q)$	real $\sqrt[q]{x}$, $q \in \mathbb{Z} \setminus \{0\}$	$\begin{cases} \mathbb{R} \text{ if } q > 0 \text{ odd} \\ [0, \infty) \text{ if } q > 0 \text{ even} \\ \mathbb{R} \setminus \{0\} \text{ if } q < 0 \text{ odd} \\ (0, \infty) \text{ if } q < 0 \text{ even} \end{cases}$	same as domain	a
$\begin{cases} \text{expm1}(x) \\ \text{exp2m1}(x) \\ \text{exp10m1}(x) \end{cases}$	$b^x - 1$	\mathbb{R}	$(-1, \infty)$	b, c
$\begin{cases} \text{logp1}(x) \\ \text{log2p1}(x) \\ \text{log10p1}(x) \end{cases}$	$\log_b(x+1)$	$(-1, \infty)$	\mathbb{R}	b, c
$\text{compoundm1}(x, y)$	$(1+x)^y - 1$	$\{x > -1\} \cup \{x = -1, y > 0\}$	$[0, \infty)$	c, d
$\text{hypot}(x, y)$	$\sqrt{x^2 + y^2}$	\mathbb{R}^2	$[0, \infty)$	
$\text{rSqrt}(x)$	$1/\sqrt{x}$	$(0, \infty)$	$(0, \infty)$	
$\text{sinPi}(x)$	$\sin(\pi x)$	\mathbb{R}	$[-1, 1]$	e
$\text{cosPi}(x)$	$\cos(\pi x)$	\mathbb{R}	$[-1, 1]$	e
$\text{tanPi}(x)$	$\tan(\pi x)$	$\mathbb{R} \setminus \{k + \frac{1}{2} \mid k \in \mathbb{Z}\}$	\mathbb{R}	e
$\text{asinPi}(x)$	$\arcsin(x)/\pi$	$[-1, 1]$	$[-1/2, 1/2]$	e
$\text{acosPi}(x)$	$\arccos(x)/\pi$	$[-1, 1]$	$[0, 1]$	e
$\text{atanPi}(x)$	$\arctan(x)/\pi$	\mathbb{R}	$(-1/2, 1/2)$	e
$\text{atan2Pi}(y, x)$	$\text{atan2}(y, x)/\pi$	$\mathbb{R}^2 \setminus \{(0, 0)\}$	$(-1, 1]$	e, f

Notes to Table 5

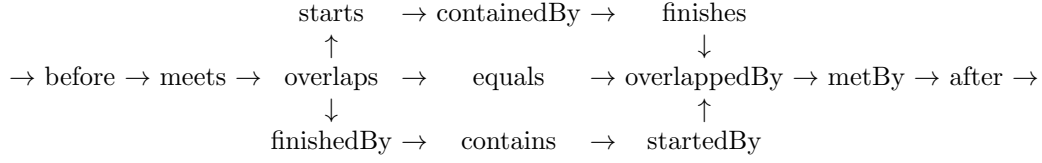
- Regarded as a family of functions parameterized by the integer arguments q , or r and s .
- $b = e, 2$ or 10 , respectively.
- Mathematically unnecessary, but included to let implementations give better numerical behavior for small values of the arguments.
- In describing domains, notation such as $\{y = 0\}$ is short for $\{(x, y) \in \mathbb{R}^2 \mid y = 0\}$, and so on.
- These functions avoid a loss of accuracy due to π being irrational, cf. Table 1, note e.
- To avoid confusion with notation for open intervals, in this table coordinates in \mathbb{R}^2 are delimited by angle brackets $\langle \rangle$.

9.7.2. Extended interval comparisons.

The **interval overlapping operation** $\text{overlap}(\mathbf{a}, \mathbf{b})$, also written $\mathbf{a} \oslash \mathbf{b}$, arises from the work of J.F. Allen [1] 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 (\mathbf{a}, \mathbf{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 $\text{overlap}(\mathbf{a}, \mathbf{b})$.

To describe the states for nonempty intervals of positive width, it is useful to think of $\mathbf{b} = [\underline{b}, \bar{b}]$ (with $\underline{b} < \bar{b}$) as fixed, while $\mathbf{a} = [\underline{a}, \bar{a}]$ (with $\underline{a} < \bar{a}$) 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 \mathbf{a} and \mathbf{b} , the value of $\mathbf{a} \oslash \mathbf{b}$ 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 $\underline{a} < \underline{b} < \bar{a} < \bar{b}$.

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 $\underline{a}, \bar{a}, \underline{b}, \bar{b}$, 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 [6]. 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.]

9.7.3. 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 \mathbf{a}, \mathbf{b} .
Notation \forall_a means “for all a in \mathbf{a} ”, and so on. Phrases within a cell are joined by “and”, e.g. **starts** is specified by $(\underline{a} = \underline{b} \wedge \bar{a} < \bar{b})$.

State $\mathbf{a} \oslash \mathbf{b}$ is	Set specification	Endpoint specification	Diagram
States with either interval empty			
bothEmpty	$\mathbf{a} = \emptyset \wedge \mathbf{b} = \emptyset$		
firstEmpty	$\mathbf{a} = \emptyset \wedge \mathbf{b} \neq \emptyset$		
secondEmpty	$\mathbf{a} \neq \emptyset \wedge \mathbf{b} = \emptyset$		
States with both intervals nonempty			
before	$\forall_a \forall_b a < b$	$\bar{a} < \underline{b}$	
meets	$\forall_a \forall_b a \leq b$ $\exists_a \forall_b a < b$ $\exists_a \exists_b a = b$	$\underline{a} < \bar{a}$ $\bar{a} = \underline{b}$ $\underline{b} < \bar{b}$	
overlaps	$\exists_a \forall_b a < b$ $\exists_b \forall_a a < b$ $\exists_a \exists_b b < a$	$\underline{a} < \underline{b}$ $\underline{b} < \bar{a}$ $\bar{a} < \bar{b}$	
starts	$\exists_a \forall_b a \leq b$ $\exists_b \forall_a b \leq a$ $\exists_b \forall_a a < b$	$\underline{a} = \underline{b}$ $\bar{a} < \bar{b}$	
containedBy	$\exists_b \forall_a b < a$ $\exists_b \forall_a a < b$	$\underline{b} < \underline{a}$ $\bar{a} < \bar{b}$	
finishes	$\exists_b \forall_a b < a$ $\exists_a \forall_b b \leq a$ $\exists_b \forall_a a \leq b$	$\underline{b} < \underline{a}$ $\bar{a} = \bar{b}$	
equal	$\forall_a \exists_b a = b$ $\forall_b \exists_a b = a$	$\underline{a} = \underline{b}$ $\bar{a} = \bar{b}$	
finishedBy	$\exists_a \forall_b a < b$ $\exists_b \forall_a a \leq b$ $\exists_a \forall_b b \leq a$	$\underline{a} < \underline{b}$ $\bar{b} = \bar{a}$	
contains	$\exists_a \forall_b a < b$ $\exists_a \forall_b b < a$	$\underline{a} < \underline{b}$ $\bar{b} < \bar{a}$	
startedBy	$\exists_b \forall_a b \leq a$ $\exists_a \forall_b a \leq b$ $\exists_a \forall_b b < a$	$\underline{b} = \underline{a}$ $\bar{b} < \bar{a}$	
overlappedBy	$\exists_b \forall_a b < a$ $\exists_a \forall_b b < a$ $\exists_b \exists_a a < b$	$\underline{b} < \underline{a}$ $\underline{a} < \bar{b}$ $\bar{b} < \bar{a}$	
metBy	$\forall_b \forall_a b \leq a$ $\exists_b \exists_a b = a$ $\exists_b \forall_a b < a$	$\underline{b} < \bar{b}$ $\bar{b} = \underline{a}$ $\underline{a} < \bar{a}$	
after	$\forall_b \forall_a b < a$	$\bar{b} < \underline{a}$	

TABLE 7. Recommended slope functions.

Name	Definition	Point function domain	Point function range	Note
expSlope1 (x)	$\frac{1}{x}(e^x - 1)$	\mathbb{R}	$(0, \infty)$	
expSlope2 (x)	$\frac{2}{x^2}(e^x - 1 - x)$	\mathbb{R}	$(0, \infty)$	
logSlope1 (x)	$\frac{2}{x^2}(\log(1+x) - x)$	\mathbb{R}	$(0, \infty)$	
logSlope2 (x)	$\frac{3}{x^3}(\log(1+x) - x + \frac{x^2}{2})$	\mathbb{R}	$(0, \infty)$	
cosSlope2 (x)	$-\frac{2}{x^2}(\cos x - 1)$	\mathbb{R}	$[0, 1]$	
sinSlope3 (x)	$-\frac{6}{x^3}(\sin x - x)$	\mathbb{R}	$(0, 1]$	
asinSlope3 (x)	$\frac{6}{x^3}(\arcsin x - x)$	$[-1, 1]$	$[1, 3\pi - 6]$	
atanSlope3 (x)	$-\frac{3}{x^3}(\arctan x - x)$	\mathbb{R}	$(0, 1]$	
coshSlope2 (x)	$\frac{2}{x^2}(\cosh x - 1)$	\mathbb{R}	$[1, \infty)$	
sinhSlope3 (x)	$\frac{3}{x^3}(\sinh x - x)$	\mathbb{R}	$[\frac{1}{2}, \infty)$	

10. The decoration system at Level 1

10.1. Decorations and decorated intervals overview. The decoration system of the set-based flavor conforms to the principles of Clause 7.1. An implementation makes the decoration system available by providing:

- a decorated version of each interval extension of an arithmetic operation, of each interval constructor, and of some other operations;
- various auxiliary functions, e.g., to extract a decorated interval’s interval and decoration parts, and to apply a standard initial decoration to an interval.

The system is specified here at a mathematical level, with the finite-precision aspects in §12. §10.2, 10.3, 10.4 give the basic concepts. §10.5, 10.6 define how intervals are given an initial decoration, and how decorations are bound to library interval operations to give correct propagation through expressions. §10.7 lists operations that do *not* propagate decorations. §10.9 discusses the decoration of user-defined arithmetic operations. §10.10 specifies the sixth decoration **com**, which is required in a multi-flavor implementation. §10.11 defines a restricted decorated interval arithmetic that suffices for some important applications and is easier to implement efficiently.

In Annex C, Clause 19 gives examples of the meaning and use of decorations; and Clause 20.1 contains a rigorous theoretical foundation, including a proof of the Fundamental Theorem of Decorated Interval Arithmetic for this flavor.

10.2. Definitions and basic properties. Formally, a decoration d is a property $p_d(f, \mathbf{x})$ of pairs (f, \mathbf{x}) , where f is a real-valued function with domain $\text{Dom}(f) \subseteq \mathbb{R}^n$ for some $n \geq 0$ and $\mathbf{x} \in \mathbb{IR}^n$ is an n -dimensional box, regarded as a subset of \mathbb{R}^n . The notation (f, \mathbf{x}) unless said otherwise denotes such a pair, for arbitrary n , f and \mathbf{x} . Equivalently and more usually, d is identified with the set of pairs for which the property holds:

$$d = \{ (f, \mathbf{x}) \mid p_d(f, \mathbf{x}) \text{ is true} \}. \quad (15)$$

This flavor provides a set \mathbb{D} of either five or six decorations. The basic five are:

Value	Short description	Property	Definition
dac	defined & continuous	$p_{\text{dac}}(f, \mathbf{x})$	\mathbf{x} is a nonempty subset of $\text{Dom}(f)$, and the restriction of f to \mathbf{x} is continuous;
def	defined	$p_{\text{def}}(f, \mathbf{x})$	\mathbf{x} is a nonempty subset of $\text{Dom}(f)$;
trv	trivial	$p_{\text{trv}}(f, \mathbf{x})$	always true (so gives no information);
emp	empty	$p_{\text{emp}}(f, \mathbf{x})$	$\mathbf{x} \cap \text{Dom}(f)$ is empty;
ill	ill-formed	$p_{\text{ill}}(f, \mathbf{x})$	$\text{Dom}(f)$ is empty.

These are listed according to the propagation order (26), which may also be thought of as a quality-order of (f, \mathbf{x}) pairs—decorations above **trv** are “good” and those below are “bad”. The sixth decoration **com** (§10.10), if provided, lies at the top of the list as “better” than these five.

A **decorated interval** is a pair, written interchangeably as (\mathbf{u}, d) or \mathbf{u}_d , where $\mathbf{u} \in \mathbb{IR}$ is a real interval and $d \in \mathbb{D}$ is a decoration. (\mathbf{u}, d) may also denote a decorated box $((\mathbf{u}_1, d_1), \dots, (\mathbf{u}_n, d_n))$, where \mathbf{u} and d are the vectors of interval parts \mathbf{u}_i and decoration parts d_i , respectively. The set of decorated intervals is denoted by \mathbb{DIR} , and the set of decorated boxes with n components is denoted by \mathbb{DIR}^n .

When several named intervals are involved, the decorations attached to $\mathbf{u}, \mathbf{v}, \dots$ are often named du, dv, \dots for readability, for instance (\mathbf{u}, du) or \mathbf{u}_{du} , etc.

An interval or decoration may be called a **bare** interval or decoration, to emphasize that it is not a decorated interval.

Treating the decorations as sets as in (15), **trv** is the set of all (f, \mathbf{x}) pairs, and the others are nonempty subsets of **trv**. By design they satisfy the **exclusivity rule**

$$\text{For any two decorations, either one contains the other or they are disjoint.} \quad (17)$$

Namely the definitions (16) give:

$$\text{dac} \subseteq \text{def} \subseteq \text{trv} \supseteq \text{emp} \supseteq \text{ill}, \quad \text{note the change from } \subseteq \text{ to } \supseteq; \quad (18)$$

$$\text{dac and def are disjoint from emp and ill.} \quad (19)$$

By (17), for any (f, \mathbf{x}) there is a unique tightest (in the containment order (18)), decoration such that $p_d(f, \mathbf{x})$ is true, called the **strongest decoration of (f, \mathbf{x})** , or of f over \mathbf{x} , and written $\text{dec}(f, \mathbf{x})$. That is:

$$\text{dec}(f, \mathbf{x}) = d \iff p_d(f, \mathbf{x}) \text{ holds, but } p_e(f, \mathbf{x}) \text{ fails for all } e \subset d. \quad (20)$$

[*Note. Like the exact range $\text{Rge}(f | \mathbf{x})$, the strongest decoration is a theoretically well-defined value, but to compute it in a program for a particular f and \mathbf{x} may be impractically expensive, or even undecidable.*]

10.3. Ill-formed intervals. The “ill-formed” decoration `ill` propagates unconditionally through arithmetic expressions. Namely, the decorated interval result of a library arithmetic operation is ill-formed (decorated with `ill`) if and only if one of its inputs is ill-formed.

Ill-formed decorated intervals result from an invalid constructor call and also may be produced in any context where an implementation determines, statically or dynamically, that it is to do interval-evaluation of an expression that, as a point function, is nowhere-defined. An ill-formed decorated interval is also called **NaI, Not an Interval**. Generally, an implementation behaves as if there is only one NaI, whose interval part is indeterminate. However, the `intervalPart()` operation must return a bare interval for any decorated interval input, and for NaI this shall be Empty. Thus one may regard NaI as being \emptyset_{ill} .

An exception is that other information may be stored in an NaI in an implementation-defined way (like the payload of a 754 floating point NaN), and functions may be provided for a user to read this for diagnostic purposes.

[*Examples. The constructor call `nums2interval(2, 1)` is invalid in this flavor, so its decorated version returns NaI. A compiler might determine that the constant expression $[1, 1]/[0, 0]$ comes from the undefined constant (zero-argument point function) $1/0$, so that it returns NaI. With more work, a compiler might determine that interval-evaluation of $\sqrt{-1 - x^2}$ should give NaI, for any input interval, because it is everywhere undefined as a real point function.*]

10.4. Permitted combinations. A decorated interval \mathbf{y}_{dy} shall always be such that $\mathbf{y} \supseteq \text{Rge}(f | \mathbf{x})$ and $p_{dy}(f, \mathbf{x})$ holds, for some (f, \mathbf{x}) as in §10.2—informally, it must tell the truth about some conceivable evaluation of a function over a box. If $dy = \text{dac}$ or `def` then by definition \mathbf{x} is nonempty, and f is everywhere defined on it, so that $\text{Rge}(f | \mathbf{x})$ is nonempty, implying \mathbf{y} is nonempty. Hence the decorated intervals \emptyset_{dac} and \emptyset_{def} , and \emptyset_{com} if `com` is provided, are contradictory: implementations shall not produce them.

No other combinations are essentially forbidden. However it is a consequence of a Level 2 requirement that the result \mathbf{y}_{dy} of a constructor or a library arithmetic operation has $dy = \text{emp}$ or `ill` if and only if \mathbf{y} is empty. Thus \emptyset_{trv} , and \mathbf{y}_{emp} or \mathbf{y}_{ill} with nonempty \mathbf{y} , are not generated by arithmetic expressions when initialized according to §10.5 and evaluated according to §10.6. However, they may be created by other means.

10.5. Initial decoration. Correct use of decorations when evaluating an expression has two parts: correctly initialize the input intervals; and evaluate using decorated interval extensions of library operations. To provide correct initialization, the function `newDec()` is provided. For a single bare interval \mathbf{x} , `newDec()` decorates it with $\text{dec}(\text{Id}, \mathbf{x})$, the strongest decoration d_x that makes $p_{d_x}(\text{Id}, \mathbf{x})$ true, where Id is the identity function $\text{Id}(x) = x$ for real x . That is,

$$\text{newDec}(\mathbf{x}) = \mathbf{x}_d \quad \text{where} \quad d = \text{dec}(\text{Id}, \mathbf{x}) = \begin{cases} \text{dac} & \text{if } \mathbf{x} \text{ is nonempty,} \\ \text{emp} & \text{if } \mathbf{x} \text{ is empty,} \end{cases} \quad (21)$$

see §10.10 for the change to this if the `com` decoration is provided. For an already decorated interval \mathbf{x}_{dx} , `newDec()` discards the decoration and acts on the interval part,

$$\text{newDec}(\mathbf{x}_{dx}) = \text{newDec}(\mathbf{x}). \quad (22)$$

For a vector of n bare or decorated intervals, `newDec()` acts componentwise to give a vector of n decorated intervals.

In this document a bare interval constant, in a context that expects a decorated interval, is implicitly *promoted* to a decorated interval by applying the `newDec` function. Whether an implementation provides such promotion at the program level is language-defined.

[Example. \emptyset is promoted to $\text{newDec}(\emptyset) = \emptyset_{\text{emp}}$, and $[1, +\infty]$ to $\text{newDec}([1, +\infty]) = [1, +\infty]_{\text{dac}}$; while $[1, 2]$ is promoted to $[1, 2]_{\text{dac}}$ if com is not provided, but $[1, 2]_{\text{com}}$ if it is.]

10.6. Decorations and arithmetic operations. Given a scalar point function φ of k variables, a **decorated interval extension** of φ —denoted here by the same name φ —adds a decoration component to a bare interval extension of φ . It has the form $\mathbf{w}_{dw} = \varphi(\mathbf{v}_{dv})$, where $\mathbf{v}_{dv} = (\mathbf{v}, dv)$ is a k -component decorated box $((\mathbf{v}_1, dv_1), \dots, (\mathbf{v}_k, dv_k))$. By the definition of a bare interval extension, the interval part \mathbf{w} depends only on the input intervals \mathbf{v} ; the decoration part dw generally depends on both \mathbf{v} and dv . In this context, Nal is regarded as being \emptyset_{ill} .

The definition of a bare interval extension implies

$$\mathbf{w} \supseteq \text{Rge}(\varphi | \mathbf{v}), \quad (\text{enclosure}). \quad (23)$$

The decorated interval extension of φ determines a dv_0 such that

$$p_{dv_0}(\varphi, \mathbf{v}) \text{ holds,} \quad (\text{a “local decoration”}). \quad (24)$$

It then evaluates the output decoration dw by

$$dw = \min\{dv_0, dv_1, \dots, dv_k\}, \quad (\text{the “min-rule”}), \quad (25)$$

where the minimum is taken with respect to the **propagation order**:

$$\text{dac} > \text{def} > \text{trv} > \text{emp} > \text{ill}. \quad (26)$$

[Notes.

1. Because Nal is treated as \emptyset_{ill} , this definition implies (without treating it as a special case) that $\varphi(\mathbf{v}_{dv})$ is Nal if, and only if, some component of \mathbf{v}_{dv} is Nal .
2. Let $f(z_1, \dots, z_n)$ be an expression defining a real point function $f(x_1, \dots, x_n)$. Then decorated interval evaluation of f on a correctly initialized input decorated box \mathbf{x}_{dx} gives a decorated interval \mathbf{y}_{dy} such that not only, by the Fundamental Theorem of Interval Arithmetic, one has

$$\mathbf{y} \supseteq \text{Rge}(f | \mathbf{x}) \quad (27)$$

but also

$$p_{dy}(f, \mathbf{x}) \text{ holds.} \quad (28)$$

For instance, if the computed dy equals def then f is proven to be everywhere defined on the box \mathbf{x} . This is the **Fundamental Theorem of Decorated Interval Arithmetic (FTDIA)**. The rules for initializing and propagating decorations are key to its validity. They are justified, and a formal statement and proof of the FTDIA given, in Annex C.

Briefly, (21) gives the correct result for the simplest expression of all, where f is the identity $f(x) = x$, which contains no arithmetic operations. The decorations are designed so that the min-rule (25) embodies basic facts of set theory and analysis, such as “If each of a set of functions is everywhere defined [resp. continuous] on its input, their composition has the same property” and “If any of a set of functions is nowhere defined on its input, their composition has the same property”. It causes correct propagation of decorations through each arithmetic operation, and hence through a whole expression.

3. In the same way as the enclosure requirement (23) is compatible with many bare interval extensions, typically coming from different interval types at Level 2, so there may be several dv_0 satisfying the local decoration requirement (24). The ideal choice is the strongest decoration d such that $p_d(\varphi, \mathbf{v})$ holds, that is to take

$$dv_0 = \text{dec}(\varphi, \mathbf{v}). \quad (29)$$

This is easily computable in finite precision for the arithmetic operations in §9.6, 9.7—see the tables in Annex C, Clause 18. However, functions may be added to the library in future for which (29) is impractical to compute for some arguments \mathbf{v} . Hence the weaker requirement (24) is made.

]

10.7. Non-arithmetic operations.

The following give interval results but are not interval extensions of point functions:

- The reverse-mode operations of §9.6.5.
- The cancellative operations `cancelPlus(x, y)` and `cancelMinus(x, y)` of §9.6.6.
- The set-oriented operations `intersection(x, y)` and `convexHull(x, y)` of §9.6.7.

The decorated interval version of each such operation distinguishes NaI inputs but otherwise returns a decoration indicating no information, as follows. If any input is NaI, the result is NaI. Otherwise the result is obtained by applying the corresponding operation to the interval parts of the inputs, and decorating its result with `trv`.

The user is responsible for applying a more informative decoration to the result via `setDec()`, where this can be deduced in a given context. Other versions of the operations may be provided in a language- or implementation-defined way, that apply such a context-adapted decoration. In particular, to simplify defining functions piecewise, an implementation may provide:

`intersectionDec(xdx, ydy)` is as `intersection(xdx, ydy)`, except that it decorates the result with `min(dx, dy)`.

`convexHullDec(xdx, ydy)` is as `convexHull(xdx, ydy)`, except that it decorates the result with the tightest (in the containment order (18)) decoration that contains both `dx` and `dy`.

A language may make these operations its default operations for `intersection` and `convexHull`. If it does so, its documentation shall warn that misuse might lead to violation of the FTDIA.

The following give non-interval results:

- The numeric functions of §9.6.9.
- The comparisons and other boolean-valued functions of §9.6.10.
- The overlap function of §9.7.2.

If any input is NaI, the result is undefined at Level 1. Otherwise, each such operation acts on decorated intervals by discarding the decoration and applying the corresponding bare interval operation.

10.8. Disassembling and assembling decorated intervals. An implementation shall provide the following operations. [Note. Careless use of the `setDec` function can negate the aims of the decoration system and lead to false conclusions that violate the FTDIA. It is provided for expert users, who may need it, e.g., to decorate the output of functions whose definition involves the `intersection` and `convexHull` operations.]

Given a decorated interval `xdx`, the operations `intervalPart(xdx)` and `decorationPart(xdx)` return `x` and `dx` respectively.

Given an interval `x` and a decoration `dx`, the operation `setDec(x, dx)` returns the decorated interval `xdx`. If this would produce one of the forbidden combinations—that is, `x = ∅` and `dx` is one of `def`, `dac` or `com`—then `NaI = ∅ill` is returned instead.

For these operations NaI is deemed to equal `∅ill`, so that `intervalPart(NaI)` returns `∅` and `decorationPart(NaI)` returns `ill`; while `setDec(x, ill)` returns NaI for any interval `x`, whether empty or not.

In implementations that do not support the `com` decoration, `com` shall still be a valid second argument to `setDec`, and give a result as if `dac` had been given.

10.9. User-supplied functions. A user may define a decorated interval extension of some point function, as defined in §10.6, to be used within expressions as if it were a library operation. [Examples.

(i) In an application, an interval extension of the function

$$f(x) = x + 1/x$$

was required. As it stands it gives unnecessarily pessimistic enclosures: e.g., with $x = [\frac{1}{2}, 2]$, one obtains

$$f(x) = [\frac{1}{2}, 2] + 1/[\frac{1}{2}, 2] = [\frac{1}{2}, 2] + [\frac{1}{2}, 2] = [1, 4],$$

much wider than $\text{Rge}(f \mid x) = [2, 2\frac{1}{2}]$.

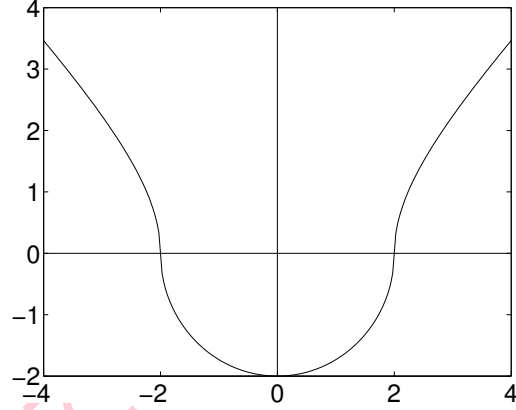
Thus it is useful to code a tight interval extension by special methods, e.g. monotonicity arguments, and provide this as a new library function. Suppose this has been done. To convert it to a decorated interval extension just entails adding code to provide a local decoration and combine this with the input decoration by the min-rule (25). In this case it is straightforward to compute the strongest local decoration $d = \text{dec}(f, x)$, as follows.

$$d = \begin{cases} \text{emp} & \text{if } x = \emptyset \text{ or } x = [0, 0], \\ \text{trv} & \text{if } 0 \in x \neq [0, 0], \\ \text{dac} & \text{if } 0 \notin x \neq \emptyset. \end{cases}$$

(ii)

The next example shows how an expert may manipulate decorations explicitly to give a function, defined piecewise by different formulas in different regions of its domain, the best possible decoration. Suppose that

$$f(x) = \begin{cases} f_1(x) := \sqrt{x^2 - 4} & \text{if } |x| > 2, \\ f_2(x) := -\sqrt{4 - x^2} & \text{otherwise,} \end{cases}$$



see the diagram.

The function consists of three pieces, on regions $x \leq -2$, $-2 \leq x \leq 2$ and $x \geq 2$, that join continuously at region boundaries, but the standard gives no way to determine this continuity, at run time or otherwise. For instance, if f is implemented by the case function, the continuity information is lost when evaluating it on, say, $x = [1, 3]$, where both branches contribute for different values of $x \in x$.

However, a user-defined decorated interval function as defined below provides the best possible decorations.

```
function  $y_{dy} = f(x_{dx})$ 
 $u = f_1(x \cap [-\infty, -2])$ 
 $v = f_2(x \cap [-2, 2])$ 
 $w = f_1(x \cap [2, +\infty])$ 
 $y = \text{convexHull}(\text{convexHull}(u, v), w)$ 
 $dy = dx$ 
```

The user's knowledge that f is everywhere defined and continuous is expressed by the statement $dy = dx$, propagating the input decoration unchanged. f , thus defined, can safely be used within a larger decorated interval evaluation.

]

10.10. The com decoration. A multi-flavor implementation shall, and other implementations may, provide the sixth decoration **com** (see §7.3 in Chapter 1), namely:

Value	Short description	Property	Definition	(30)
com	common	$p_{\text{com}}(f, x)$	x is a bounded, nonempty subset of $\text{Dom}(f)$; f is continuous at each point of x ; and the computed interval $f(x)$ is bounded.	

Its position in the containment and propagation orders is given by

$$\text{com} \subseteq \text{dac} \subseteq \text{def} \subseteq \text{trv} \supseteq \text{emp} \supseteq \text{ill},$$

$$\text{com} > \text{dac} > \text{def} > \text{trv} > \text{emp} > \text{ill}.$$

Including **com** causes the following changes to the decoration system:

- The strongest decoration of the identity function, $\text{dec}(\text{Id}, x)$ now equals **com** when x is bounded and nonempty.

- Hence the **newDec** function gives the decoration **com** instead of **dac** to a bounded, nonempty interval, namely

$$\text{newDec}(x) = x_d \quad \text{where} \quad d = \text{dec}(\text{Id}, x) = \begin{cases} \text{com} & \text{if } x \text{ is bounded and nonempty,} \\ \text{dac} & \text{if } x \text{ is unbounded,} \\ \text{emp} & \text{if } x \text{ is empty.} \end{cases} \quad (31)$$

- Each arithmetic operation gives **com** as its local decoration if the conditions (30) are satisfied. The propagation rule specified by (23, 24, 25) is unchanged. Note that minor changes are needed to the first example in §10.9.

[Notes.

- The **com** decoration describes both a Level 1 and a Level 2 property. When f is a library arithmetic operation φ , the computed interval $\varphi(x)$ means the enclosure of $\text{Rge}(\varphi | x)$ computed by a particular Level 2 interval extension of φ , giving a result of some interval type \mathbb{T} . Thus $p_{\text{com}}(\varphi, x)$ indicates a finite-precision evaluation that is common in the flavor sense, giving a bounded enclosure of the range.
- When f is a function defined by an expression, the computed interval $f(x)$ means the enclosure y of $\text{Rge}(f | x)$ produced by interval evaluation of the expression at Level 2. Evaluation need not use just one interval type \mathbb{T} : any combination of supported types is permitted. If the final y is decorated with **com**, it follows from the propagation rules that the whole evaluation was common. That is, the final enclosure of the range is bounded, as were all the intermediate intervals. In addition, the function f is everywhere continuous on the bounded, nonempty input box x .
- When **com** is provided, it is possible to detect overflow (at Level 2) during interval evaluation $f(x)$ of an arithmetic expression, in the following limited sense. Define overflow to be the event that the result of a library interval operation is mathematically a bounded set, but could not be enclosed in any bounded interval of the destination interval type—because none exists, or possibly because it is too expensive for the implementation to determine if one exists or not.

Suppose x is a bounded nonempty box, so initially its components are all decorated **com**. If the final result y is decorated **dac**, it follows that the point function f is everywhere defined and continuous on x and hence, by a compactness argument, is mathematically bounded on x . However, an unbounded result-interval must have occurred at some stage of evaluating $f(x)$, otherwise the final decoration would have been **com**. That is, a final decoration **dac** rather than **com** is diagnostic of overflow. However, if the result decoration is **def** or **trv**, which are the other possibilities in this context, then nothing can be concluded about overflow.

]

10.11. Compressed arithmetic with a threshold.

The **compressed decorated interval arithmetic** (compressed arithmetic for short) described here lets experienced users obtain more efficient execution in applications where the use of decorations is limited to the context described below. An implementation need not provide it; if it does so, the behavior described in this subclause is required.

Each Level 2 instance of compressed arithmetic is based on a supported Level 2 bare interval type \mathbb{T} , but is a distinct **compressed type** derived from its **parent type** \mathbb{T} , with its own objects and library of operations. Conversions are provided between a compressed type and its parent type.

Compressed arithmetic uses the standard set of 5 or 6 decorations (16). The context is that, frequently, the use that is made of a decorated interval function evaluation $y_{dy} = f(x_{dx})$ depends on a check of the result decoration dy against an application-dependent **exception threshold** τ , where $\tau \geq \text{trv}$ in the propagation order (26):

- $dy \geq \tau$ represents normal computation. The decoration is not used, but one exploits the range enclosure given by the interval part and the knowledge that dy remained $\geq \tau$.
- $dy < \tau$ declares an exception to have occurred. The interval part is not used, but one exploits the information given by the decoration.

For such uses, one needs to store an interval's value, or its decoration, but never both at once. A **compressed interval** is an object whose value is either a bare interval (of the parent type), or a bare decoration. The values are arbitrary except that:

- The empty interval is not used: the decoration `emp` or `ill` is used instead.
- Only decorations $< \tau$ occur, so `com` is never used; if `com` is not provided by the implementation then `dac` is never used.

At Level 2, different thresholds generate different compressed interval types. That is, if \mathbb{T} is a parent type for compressed arithmetic, there shall be separate compressed interval types \mathbb{T}_τ for each threshold value $\tau \geq \text{trv}$. The only way to use compressed arithmetic with a particular threshold τ is to construct \mathbb{T}_τ -intervals, that is, objects of type \mathbb{T}_τ .

[*Note. Since, for any practical interval type \mathbb{T} , a decoration fits into less space than an interval, one can implement arithmetic on “compressed interval” objects that take up the same space as a bare interval of that type. For instance if \mathbb{T} is the IEEE754 binary64 inf-sup type, a compressed interval uses 16 bytes, the same as a bare \mathbb{T} -interval; a full decorated \mathbb{T} -interval needs at least 17 bytes.*]

Because compressed intervals must behave exactly like bare intervals as long as one does not fall below the threshold, and take up the same space, there is no room to encode τ as part of the interval’s value. “Mixed threshold” operations, combining compressed intervals of the same parent type and different threshold values, can be done in effect by first converting the input operands to the destination type, as described below. It is the user’s responsibility to ensure that this is valid in the context of the application.]

A program may use compressed arithmetic with more than one threshold. Conversion between thresholds, say from a \mathbb{T}_σ -interval to a \mathbb{T}_τ -interval, shall be equivalent to first converting to a normal decorated interval by `normalInterval()`, and then to the destination type by `τ -compressedInterval(\mathbf{X})`. Such conversions need not be provided as single operations.

The enquiry function `isInterval(x)` returns true if the compressed interval x is an interval, false if it is a decoration.

The constructor `τ -compressedInterval()` is provided for each threshold value τ . The result of `τ -compressedInterval(\mathbf{X})`, where $\mathbf{X} = (x, dx)$ is a decorated interval of the parent type, is a \mathbb{T}_τ -interval as follows:

```
if  $dx \geq \tau$ , return the  $\mathbb{T}_\tau$ -interval with value  $x$ 
else return the  $\mathbb{T}_\tau$ -interval with value  $dx$ .
```

`τ -compressedInterval(x)` for a bare interval x is equivalent to `τ -compressedInterval(newDec(x))`.

The function `normalInterval(x)` converts a \mathbb{T}_τ -interval to a decorated interval of the parent type, as follows:

```
if  $x$  is an interval, return  $(x, \tau)$ .
if  $x$  is a decoration  $d$ 
    if  $d$  is ill or emp, return (Empty,  $d$ )
    else return (Entire,  $d$ ).
```

Operations, whether arithmetic or not, on compressed intervals shall behave as if the following is done:

1. Each compressed interval argument is converted to a decorated interval by `normalInterval`;
2. the corresponding operation of the parent decorated interval type is performed;
3. the result, if an interval, is converted back to a compressed interval by `τ -compressedInterval`.

(32)

Since there are only a few decorations, one can prepare complete operation tables according to this rule, and only these tables need to be implemented. Sample tables and worked examples are given in §23 in Annex C.

[*Note. It can be shown, see §20.2 for details, that for arithmetic operations, the behavior (32) derives from that for normal decorated interval operations, following worst case semantics rules that treat a decoration in $\{\text{trv}, \text{def}, \text{dac}\}$ as representing a set of decorated intervals, and are necessary if the fundamental theorem is to remain valid. Namely, inputs to each operation behave as follows:*

- Operations purely on bare intervals are performed as if each x is the decorated interval x_τ , resulting in a decorated interval y_{dy} that is then converted back into a compressed interval. If $dy < \tau$, the result is the bare decoration dy , otherwise the bare interval y .

- For operations with at least one bare decoration input, the result is always a bare decoration. A bare interval input is treated as in item (a). A decoration d in $\{\text{emp}, \text{ill}\}$ is treated as \emptyset_d . A decoration d in $\{\text{trv}, \text{def}, \text{dac}\}$ is treated (conceptually) as an arbitrary x_d with nonempty interval x . The decoration com cannot occur. Performing the resulting decorated interval operation on all such possible inputs leads to a set of all possible results y_{dy} . The tightest decoration (in the containment order (18)) enclosing all resulting dy is returned.

As a result each operation returns an actual or implied decoration compatible with its input, so that in an extended evaluation, the final decoration using compressed arithmetic is never stronger than that produced by full decorated interval arithmetic.

For instance, assuming $\tau > \text{def}$,

- (a) $\text{def}/[1, 2]$ becomes $x_{\text{def}}/[1, 2]_\tau$ with arbitrary nonempty interval x ; the result is always decorated def , so returns def .
- (b) But $[1, 2]/\text{def}$ becomes $[1, 2]_\tau/x_{\text{def}}$ with arbitrary nonempty interval x ; The result can be decorated def , trv or emp , so returns the tightest decoration containing these, namely trv .

If compressed arithmetic is implemented, it shall provide versions of all the required operations of §9.6, and it should provide the recommended operations of §9.7.

DRAFT 7.0

11. Level 2 description

11.1. 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**¹ 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 \mathbb{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 §11.6, represent finite sets of (mathematical, Level 1) intervals.
- **Decorated interval types**, see §12, represent finite sets of decorated intervals.
- **Compressed interval types** (optional), see §12.5, 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 §11.6.2, of at least one of the five basic formats of 754§3.3.

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, see §12.2. Beyond this, which types are supported is language- or implementation-defined.

This standard uses the term **T-version** of an operation, where \mathbb{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 \mathbb{T} -intervals. This includes the following:

- (a) A \mathbb{T} -interval extension (§11.9) of one of the required or recommended arithmetic operations of §9.6, 9.7.
- (b) A set operation, such as intersection and convex hull of \mathbb{T} -intervals, returning a \mathbb{T} -interval.
- (c) A function such as the midpoint, whose input is a \mathbb{T} -interval and output is a numeric value.
- (d) A constructor, whose input is numeric or text and output is a \mathbb{T} -interval.
- (e) The \mathbb{T} -interval hull, regarded as a conversion operation, see §11.8.3.

Generically these comprise the **operations of the type \mathbb{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.

11.2. 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-inf-sup-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 \mathbb{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.

11.3. 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 8.

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.

¹Not “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.

11.3.1. *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.

11.4. Tagging, and the meaning of equality at Level 2. 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:

$$\begin{aligned}\text{number datum} &= (\text{Level 1 number, format name}), \\ \text{interval datum} &= (\text{Level 1 interval, type name}),\end{aligned}$$

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. It follows that distinct formats or types are disjoint sets

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.*]

In this document the five basic formats of 754§3.3 are named `binary32`, `binary64`, `binary128`, `decimal64`, `decimal128`. Their inf-sup types (§11.6.2) are $\bar{\mathbb{I}}(\text{binary32})$, \dots , $\bar{\mathbb{I}}(\text{decimal128})$, the parentheses being optional. Also, abbreviated names such as `b64` instead of `binary64` are sometimes used for convenience, and refer to the same format, or type derived from it.

An interval datum may have more than one representation at Level 3 (e.g., in an inf-sup type, a zero endpoint might be stored as either -0 or $+0$). Independent of representation, two bare interval datums are defined to be equal if and only if they represent the same Level 1 interval tagged by the same or equivalent name. In particular two datums of the same bare interval type are equal if and only if they represent the same Level 1 interval.

[*Note. Two bare interval datums a, b of the same type are equal if and only if $\text{isEqual}(a, b)$ returns true. However, in the context of 754-conforming types, intervals of different types can compare equal. E.g., let a, b be the `binary32` inf-sup and `binary64` inf-sup extensions, respectively, of an interval that is represented exactly in both types, such as $[1, 2]$. Then $\text{isEqual}(a, b)$ is true, but a and b are not equal in the sense of this subclause.*]

⚠ The next passage may be controversial, and certainly needs stating more precisely. Discussion please!

Implementations shall ensure that different Level 3 representations of an interval datum cannot affect the execution of a program by conforming to the

Equality principle. Let $y = f(x_1, x_2, \dots, x_n)$ be an arbitrary evaluation at Level 2 of a predicate (boolean valued expression), where the arguments x_j can be any mix of intervals and numbers. Then y shall be unchanged if any *interval* argument x_j is replaced by an argument x'_j that equals x_j in the Level 2 sense.

Here f is in general a mixed interval/numeric expression. It is built up of library functions φ_i , which can be arithmetic operations, constructors, comparisons, or numeric functions of intervals, or point arithmetic operations or comparisons of the underlying floating point system.

Evaluation at Level 2 means that the inputs x_j are of specified types (using this to mean both interval types and number formats); and each φ_i has specified types for its inputs and output, such that each value that is used as an input is of the expected type.

[*Example. Let \mathbb{F} be a 754 format and $\mathbb{T} = \mathbb{IF}$ the derived inf-sup type. Suppose a \mathbb{T} -interval $[l, u]$ is represented at Level 3 as the pair of \mathbb{F} -numbers (l, u) . Let f be the expression*

$$1/\inf(x) > 0.$$

and consider $[0, 1]$ with the two Level 3 representations $x = (-0, 1)$ and $x' = (+0, 1)$. Then $x = x'$ in the Level 2 sense, but a naive implementation gives

$$\begin{aligned} f(x) &= \left(\frac{1}{\inf(x)} > 0 \right) &= \left(\frac{1}{-0} > 0 \right) &= (-\infty > 0) = \text{false}; \\ f(x') &= \left(\frac{1}{\inf(x')} > 0 \right) &= \left(\frac{1}{+0} > 0 \right) &= (+\infty > 0) = \text{true}. \end{aligned}$$

The standard does not say which of these two results is “correct”. But since they differ, the equality principle is violated and such an implementation is non-conforming. One way to make it conforming is to canonicalize all operations that output an interval, to ensure that for instance all zero bounds are stored as +0.]


11.5. Number formats. Having regard to §11.4, a **number format**, or just **format**, is the set of all pairs (x, f) where x belongs to a set \mathbb{F} , and f is a name for the format.

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

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

 **⚠ I deviate from motion 33 here in not allowing for $-0, +0$ to be distinct datums, hoping we can avoid this.**

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

Each format \mathbb{F} shall have an associated **rounding function** that maps any extended real number x to an element of \mathbb{F} . A constraint can be given on the rounding direction, e.g. one of the rounding direction attributes in 754§4.3. The rounded result is an element of \mathbb{F} that is closest to x subject to this constraint, with an implementation-defined rule for the distance to an infinity, and for the method of tie-breaking when more than one member of \mathbb{F} has the “closest” property.

11.6. Bare interval types.

11.6.1. *Definition.* Having regard to §11.4, a **bare interval type** is the set of all pairs (x, t) where x belongs to a finite subset \mathbb{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 \mathbb{T} , and one may say a bare interval type is an arbitrary finite set \mathbb{T} of intervals that contains Empty and Entire.

A **\mathbb{T} -interval** means an interval belonging to \mathbb{T} ; a **\mathbb{T} -box** means a box with \mathbb{T} -interval components.

[*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.]

11.6.2. Inf-sup and mid-rad types. The **inf-sup type** derived from a given number format \mathbb{F} (the type \mathbb{F} **inf-sup**, e.g., “`binary64 inf-sup`”) is the bare interval type \mathbb{T} comprising all intervals whose endpoints are in \mathbb{F} , together with Empty. \mathbb{F} is termed the **parent format** of \mathbb{T} . Note that Entire is in \mathbb{T} because $\pm\infty \in \mathbb{F}$ by the definition of a number format, so \mathbb{T} satisfies the requirements for a bare interval type given in §11.6.1.

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 \mathbb{F} , and r is in a possibly different number format \mathbb{F}' , with m, r finite and $r \geq 0$. From the definition in §11.6.1 such a type, to be conforming, must contain Empty and Entire, so at Level 3 it must have representations of these; it may also have representations of semi-bounded intervals.

11.7. 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 \mathbb{F}_n of numbers representable at the n th level ($n = 1, 2, 3, \dots$), and $\mathbb{F}_1 \subset \mathbb{F}_2 \subset \mathbb{F}_3 \dots$. These are typically used to define a corresponding infinite sequence of interval types \mathbb{T}_n with $\mathbb{T}_1 \subset \mathbb{T}_2 \subset \mathbb{T}_3 \dots$.

[*Example. For multi-precision systems that define a nonempty \mathbb{T}_n -interval to be one whose endpoints are \mathbb{F}_n -numbers, each \mathbb{T}_n is an inf-sup type with a unique interval hull operation—explicit, in the sense of §11.8.*]

A conforming implementation must define such \mathbb{T}_n as a *parameterized sequence* of interval types. It cannot take the union over n of the sets \mathbb{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.

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

11.8.1. Hull in one dimension. Each bare interval type \mathbb{T} shall have an **interval hull** operation

$$\mathbf{y} = \text{hull}_{\mathbb{T}}(\mathbf{s})$$

specified, which is part of its mathematical definition. It maps an arbitrary set of reals, \mathbf{s} , to a minimal \mathbb{T} -interval \mathbf{y} enclosing \mathbf{s} . Minimal is in the sense that

$$\mathbf{s} \subseteq \mathbf{y}, \text{ and for any other } \mathbb{T}\text{-interval } \mathbf{z}, \text{ if } \mathbf{s} \subseteq \mathbf{z} \subseteq \mathbf{y} \text{ then } \mathbf{z} = \mathbf{y}.$$

For clarity when needed, this operation is called the \mathbb{T} -hull and denoted $\text{hull}_{\mathbb{T}}$.

Since \mathbb{T} is a finite set and contains Entire, such a minimal \mathbf{y} exists for any \mathbf{s} . In general \mathbf{y} may not be unique. If it is unique for every subset \mathbf{s} of \mathbb{R} , then the type \mathbb{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. 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 \mathbb{T} be the mid-rad type (§11.6.2) where m and r use the same floating point format \mathbb{F} , say `binary64`, and let \mathbf{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 \mathbf{s} has $r = 1 + \epsilon$. But m can be any of the many \mathbb{F} -numbers in the range 0 to ϵ ; each of these gives a minimal enclosure of \mathbf{s} .

A possible general algorithm, for a bounded set \mathbf{s} and a mid-rad type, is to choose $m \in \mathbb{F}$ as close as possible to the mathematical midpoint of the interval $[\underline{s}, \bar{s}] = [\inf \mathbf{s}, \sup \mathbf{s}]$ (with some way to resolve ties) and then the smallest $r \in \mathbb{F}'$ such that $r \geq \max(m - \underline{s}, \bar{s} - m)$. The cost of performing this depends on how the set \mathbf{s} is represented. If \mathbf{s} is a `binary64 inf-sup` interval, it is simple. If \mathbf{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

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

11.8.2. *Hull in several dimensions.* In n dimensions the \mathbb{T} -hull, as defined mathematically in §11.8.1, is extended to act componentwise, namely for an arbitrary subset \mathbf{s} of \mathbb{R}^n it is $\text{hull}_{\mathbb{T}}(\mathbf{s}) = (\mathbf{y}_1, \dots, \mathbf{y}_n)$ where

$$\mathbf{y}_i = \text{hull}_{\mathbb{T}}(s_i),$$

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

11.8.3. *Interval type conversion.* For each supported bare interval type \mathbb{T} , an implementation shall provide the hull operation $\text{hull}_{\mathbb{T}}(\mathbf{x})$ for input intervals \mathbf{x} of any supported bare interval type. Conversion of an interval from one type to another, whether explicit or implicit, shall be done by applying the hull operation of the destination type.

11.9. **Level 2 interval extensions.** Let \mathbb{T} be a bare interval type and f an n -variable scalar point function. A **\mathbb{T} -interval extension** of f , also called a **\mathbb{T} -version** of f , is a mapping \mathbf{f} from n -dimensional \mathbb{T} -boxes to \mathbb{T} -intervals, that is $\mathbf{f} : \mathbb{T}^n \rightarrow \mathbb{T}$, such that $f(x) \in \mathbf{f}(\mathbf{x})$ whenever $x \in \mathbf{x}$ and $f(x)$ is defined. Equivalently

$$\mathbf{f}(\mathbf{x}) \supseteq \text{Rge}(f \mid \mathbf{x}). \quad (33)$$

for any \mathbb{T} -box $\mathbf{x} \in \mathbb{T}^n$, regarding \mathbf{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 (§9.4.3) so that this document does not distinguish between Level 2 and Level 1 extensions. Namely define \mathbf{f}^* by

$$\mathbf{f}^*(\mathbf{s}) := \mathbf{f}(\text{hull}_{\mathbb{T}}(\mathbf{s}))$$

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

In the context of a 754-conforming implementation, *typeOf* operations occur, where the inputs may be a mix of related types (e.g., the inf-sup types of `binary32` and `binary64`). In such cases, \mathbb{T} is deemed to be the union of the individual types involved.

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

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

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

[*Note. That is, widen moves each finite endpoint of \mathbf{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 \mathbb{T} -box $\mathbf{x} = (x_1, \dots, x_n)$, this function acts componentwise to produce the \mathbb{T} -box

$$\text{widen}(\mathbf{x}) = (\text{widen}(x_1), \dots, \text{widen}(x_n)).$$

For a \mathbb{T} -interval extension \mathbf{f} of an n -variable scalar point function f , the standard specifies three **accuracy modes** for \mathbf{f} :

Tightest: $\mathbf{f}(\mathbf{x})$ shall equal $\text{hull}_{\mathbb{T}}(\text{Rge}(f \mid \mathbf{x}))$, for any \mathbb{T} -box \mathbf{x} .

Accurate: $\mathbf{f}(\mathbf{x})$ shall be contained in $\text{widen}(\text{hull}_{\mathbb{T}}(\text{Rge}(f \mid \text{widen}(\mathbf{x}))))$, for any \mathbb{T} -box \mathbf{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 (33).

[*Note. In the "accurate" specification, the inner `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 outer `widen()` is to handle the problem of correct rounding, which may be hard or even undecidable for some special functions at some arguments.*]

11.11. Required operations on bare intervals.

An implementation shall provide a \mathbb{T} -version, see §11.9, of each operation listed in subclauses §11.11.1 and §11.11.3 to §11.11.10, for each supported type \mathbb{T} . That is, those of its inputs and outputs that are intervals, are of type \mathbb{T} .

A 754-conforming implementation, or part thereof, shall provide mixed-type *typeOf* operations, as specified in §11.3.1, for the following operations, which correspond to those that 754 requires to be provided as *formatOf* operations.

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 in addition to a required "tightest" one, to offer a trade-off of accuracy versus speed or code size. How such a facility is provided, is language- or implementation-defined.

11.11.1. *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 \mathbb{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 \mathbb{T} -version of the function `text2interval`, but languages may define other contexts where interval literals are converted to internal intervals in a program.

A (numeric) *floating literal* is a string that represents a real number or $\pm\infty$. If the latter, it is the string `inf` or the string `infinity`, ignoring case, optionally preceded by `+`, with value $+\infty$; or preceded by `-`, with value $-\infty$. If the former, it is a decimal or hexadecimal floating constant, of arbitrary precision, acceptable as input to the 754 standard's string-to-float conversions (see IEEE 754-2008, §5.12). Its value is the exact real number the text string represents.

A string conforming to the **interchange syntax** of interval literals 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 values they represent.

- (i) Inf-sup form: a string `[l , u]` where l and u are floating literals with $l \leq u$, $l < +\infty$ and $u > -\infty$, see §9.2. Its value is the mathematical interval $[l, u]$.
- (ii) Mid-rad form: a string `< m +- r >` where m and r are floating literals representing finite numbers with $r \geq 0$. Its value is the mathematical interval $[m - r, m + r]$.
- (iii) The strings `emp` or `empty`, ignoring case, whose value is the empty set \emptyset ; and `ent` or `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 `intervalLiteral`, subject to the constraints on l, u, m, r stated above.

floatLiteral	{floating literal in the default locale}
sp	{space character}*
infSupInterval	"[" {sp} {floatLiteral} {sp} "," {sp} {floatLiteral} {sp} "]"
midRadInterval	"<" {sp} {floatLiteral} {sp} "+-" {sp} {floatLiteral} {sp} ">"
intervalLiteral	({infSupInterval} {midRadInterval} "empty" "entire")

In an **enhanced syntax** of interval literals, which is language- or implementation-defined:

- Alternative syntax for floating literals (e.g., that of Ada) may be provided.
- **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** denoting π .
- The inf-sup style $[x]$ should be provided, with the same meaning as $[x, x]$.

[*Example. With these enhancements, for any type \mathbb{T} , applying the \mathbb{T} -version of `text2interval` to the string `[Pi]` gives a \mathbb{T} -interval enclosing π .*]

11.11.2. *Interval constants.* An implementation shall provide a \mathbb{T} -version of each constant function in §9.6.2, for each supported bare interval type \mathbb{T} . It returns a \mathbb{T} -interval.

11.11.3. *Forward-mode elementary functions.* An implementation shall provide a \mathbb{T} -version of each forward arithmetic operation in §9.6.3, for each supported bare interval type \mathbb{T} . Its inputs and output are \mathbb{T} -intervals.

For a 754-conforming type, each such operation shall have an extension of that type with accuracy mode as in Table 8(a). 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.*]

11.11.4. *Interval case expressions and case function.* An implementation shall provide the interval `case(c, g, h)` function, see §9.6.4, for each supported type \mathbb{T} . The input **c** is of an arbitrary supported interval type. The inputs **g**, **h**, and the result, are of type \mathbb{T} . The implementation shall be as if the \mathbb{T} -version of `convexHull` is used in (9) of §9.6.4.

11.11.5. *Reverse-mode elementary functions.* An implementation shall provide a \mathbb{T} -version of each reverse arithmetic operation in §9.6.5, for each supported bare interval type \mathbb{T} . Its inputs and output are \mathbb{T} -intervals.

For a 754-conforming type, each such operation shall have a version of that type with accuracy mode as in Table 8(b). For other types, the accuracy mode is language- or implementation-defined.

11.11.6. *Cancellative addition and subtraction.* An implementation shall provide a \mathbb{T} -version of each of the operations `cancelPlus` and `cancelMinus` in §9.6.6, for each supported bare interval type \mathbb{T} . Its inputs and output are \mathbb{T} -intervals.

It shall return an enclosure of the Level 1 value if the latter is defined, and Entire otherwise. In particular it shall return Empty if the Level 1 value is Empty. 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 \mathbb{T} -interval containing it.

If \mathbb{T} is a 754-conforming type, the result shall be the \mathbb{T} -hull of the Level 1 result when this is defined. [*Note. This may require computing in extra precision in boundary cases: see example in §14.6.*]

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

11.11.7. *Set operations.* An implementation shall provide a \mathbb{T} -version of each of the operations `intersection` and `convexHull` in §9.6.7, for each supported bare interval type \mathbb{T} . Its inputs and output are \mathbb{T} -intervals.

These operations shall return the \mathbb{T} -interval hull of the exact result.

[*Note. In particular, if \mathbb{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 \mathbb{T} is a 754-conforming type.*]

11.11.8. *Constructors.* There shall be a bare interval version of each constructor in §9.6.8, for each supported bare interval type \mathbb{T} . It returns a \mathbb{T} -interval.

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,

(a) Forward		(b) Reverse	
Name	Accuracy	Name	Accuracy
<code>add(x, y)</code>	tightest	<code>sqrRev(c, x)</code>	accurate
<code>sub(x, y)</code>	tightest	<code>invRev(c, x)</code>	accurate
<code>mul(x, y)</code>	tightest	<code>absRev(c, x)</code>	accurate
<code>div(x, y)</code>	tightest	<code>pownRev(c, x, p)</code>	accurate
<code>inv(x)</code>	tightest	<code>sinRev(c, x)</code>	accurate
<code>sqr(x)</code>	tightest	<code>cosRev(c, x)</code>	accurate
<code>hypot(x, y)</code>	tightest	<code>tanRev(c, x)</code>	accurate
<code>case(b, g, h)</code>	tightest	<code>coshRev(c, x)</code>	accurate
<code>sqr(x)</code>	tightest	<code>mulRev(b, c, x)</code>	accurate
<code>pown(x, p)</code>	accurate	<code>divRev1(b, c, x)</code>	accurate
<code>pow(x, y)</code>	accurate	<code>divRev2(a, c, x)</code>	accurate
<code>exp, exp2, exp10(x)</code>	tightest	<code>powRev1(b, c, x)</code>	accurate
<code>log, log2, log10(x)</code>	tightest	<code>powRev2(a, c, x)</code>	accurate
<code>sin(x)</code>	accurate	<code>atan2Rev1(b, c, x)</code>	accurate
<code>cos(x)</code>	accurate	<code>atan2Rev2(a, c, x)</code>	accurate
<code>tan(x)</code>	accurate		
<code>asin(x)</code>	accurate		
<code>acos(x)</code>	accurate		
<code>atan(x)</code>	accurate		
<code>atan2(y, x)</code>	accurate		
<code>sinh(x)</code>	accurate		
<code>cosh(x)</code>	accurate		
<code>tanh(x)</code>	accurate		
<code>asinh(x)</code>	accurate		
<code>acosh(x)</code>	accurate		
<code>atanh(x)</code>	accurate		
<code>sign(x)</code>	tightest		
<code>ceil(x)</code>	tightest		
<code>floor(x)</code>	tightest		
<code>round(x)</code>	tightest		
<code>trunc(x)</code>	tightest		
<code>abs(x)</code>	tightest		
<code>min(x_1, \dots, x_k)</code>	tightest		
<code>max(x_1, \dots, x_k)</code>	tightest		

TABLE 8. Accuracy levels for required arithmetic operations.

l and u are values of supported number formats within a program; in the latter, they are floating literals.

Evaluating b as 0 when the true value is 1 (a “false negative”) leads to falsely returning Empty as an enclosure of the true nonempty interval. 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.

A bare interval constructor call either **succeeds** or **fails**. This notion is used to determine the value returned by the corresponding decorated interval constructor. A language- or implementation-defined exception should be signaled when a bare interval constructor call fails.

[Note. 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 forbidden—the comparison $l \leq u$ requires the exact values to be used, which requires use of a version of `nums2interval` with long double arguments.]

nums2interval.

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

For all kinds, the result 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 constructor call succeeds if the implementation determines that the Level 1 value exists, or is unable to determine that it does not exist—see the discussion at the start of this subclause. In the latter case the result shall be an interval containing l and u .

The call also succeeds in the special cases $l = u = -\infty$ or $l = u = +\infty$, see below.

Otherwise the call fails, and returns $x = \emptyset$.

- An implementation shall provide at least one kind where l and u have the same format. This format should be compatible with \mathbb{T} , see §11.5.
- For \mathbb{T} belonging to a 754-conforming implementation, *formatOfkinds* shall be provided, which accept l and u having any 754 format of that implementation, of the same radix as \mathbb{T} 's parent format. The result shall be the \mathbb{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 overflowed. Therefore in this case `nums2interval` shall return the tightest \mathbb{T} -interval that is unbounded above. For any type, this exists and has the form $x = [\text{HUGEPOS}, +\infty]$, where HUGEPOS is a uniquely defined extended-real number $< +\infty$.

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

If $l = u = -\infty$, `nums2interval` shall return $[-\infty, \text{HUGENEG}]$, defined similarly.

text2interval.

Input s to the constructor `text2interval(s)` is a text string. The constructor call succeeds if the implementation determines that s is a valid interval literal with value x , see §11.11.1, and returns a \mathbb{T} -interval containing x . It also succeeds if the implementation evaluates finite bounds l, u but cannot determine that $l \leq u$. In the latter case the result shall be an interval containing l and u . Otherwise the call fails and the result is Empty.

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

11.11.9. *Numeric functions of intervals.* An implementation shall provide a \mathbb{T} -version of each numeric function in Table 3 of §9.6.9, for each supported bare interval type \mathbb{T} . It shall return a result in a supported number format \mathbb{F} as defined in §11.5. 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 \mathbb{F} is a 754-conforming format, the tie-breaking method shall follow 754§4.3.1 and 754§4.3.3; otherwise it is language- or implementation-defined.

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

- $\text{inf}(x)$ returns the Level 1 value, rounded toward $-\infty$.
- $\text{sup}(x)$ returns the Level 1 value, rounded toward $+\infty$.
- $\text{mid}(x)$ returns NaN if x is Empty, and 0 if x is Entire. Otherwise for nonempty $x = [\underline{x}, \bar{x}]$ its value is defined by the following cases.

\underline{x}, \bar{x} both finite	the closest finite \mathbb{F} -number to the Level 1 value, breaking ties appropriately;
$\underline{x} = -\infty, \bar{x}$ finite	the finite negative \mathbb{F} -number of largest magnitude;
\underline{x} finite, $\bar{x} = +\infty$	the finite positive \mathbb{F} -number of largest magnitude;

The three tabulated cases can be obtained by computing $(\underline{x} + \overline{x})/2$ exactly in the extended reals and rounding the result to the nearest finite \mathbb{F} -number.

- **rad**(x) returns NaN if x is empty, and otherwise the smallest \mathbb{F} -number r such that x is contained in the exact interval $[m - r, m + r]$, where m is the value returned by **mid**(x).
[Note. **rad**(x) may be $+\infty$ even though x is bounded, if \mathbb{F} has insufficient range. However, if \mathbb{F} is a 754 format and \mathbb{T} is the derived inf-sup type, **rad**(x) is finite for all bounded nonempty intervals.]
- **wid**(x) returns the same value as $2 * \text{rad}(x)$, rounded toward $+\infty$.
[Note. At Level 2, **wid**(x) may be infinite though **rad**(x) is finite.]
- **mag**(x) returns the Level 1 value rounded toward $+\infty$ if this value exists (if x is nonempty). Otherwise it returns NaN.
- **mig**(x) returns the Level 1 value rounded toward zero if this value exists (if x is nonempty). Otherwise it returns NaN.

11.11.10. Boolean functions of intervals.

An implementation shall provide a \mathbb{T} -version of the function **isEmpty**(x) and the function **isEntire**(x) in §9.6.10, for each supported bare interval type \mathbb{T} .

An implementation shall provide a \mathbb{T} -version of each of the comparison relations in Table 4 of §9.6.10, for each supported bare interval type \mathbb{T} . Its inputs are \mathbb{T} -intervals.

For a 754-conforming part of an implementation, mixed-type versions of these relations shall be provided, where the inputs have arbitrary 754-conforming types of the same radix.

These comparisons shall return, in all cases, the correct value of the comparison applied to the intervals represented by the inputs as if in infinite precision. In particular **isEqual** shall return **true** if and only if its arguments are equal as defined in §11.4.

11.11.11. *Complete arithmetic, dot product function.* An implementation that provides 754-conforming types shall provide **complete arithmetic**, as specified in U. Kulisch and V. Snyder [4], for the parent format \mathbb{F} of at least one such type.

This involves providing a *complete format* datatype $C(\mathbb{F})$ associated with the relevant \mathbb{F} , and associated operations. A $C(\mathbb{F})$ datum z holds a fixed-point number of the relevant radix (2 or 10), with enough digits before and after the point to let multiply-add operations $z + x * y$ be done exactly, where x and y are arbitrary finite \mathbb{F} -numbers. It also holds one bit for sign, and 3 bits for status information (equivalent to a decoration).

[Example. For the binary64 format the recommended complete format has 4 bits for sign and status, 2134 bits before the point, and 2150 after the point, for a total of 4288 bits or 536 bytes; this allows for at least 2^{88} multiply-adds before overflow can occur.]

The following operations shall be provided, see [4] for details.

- **convert** converts from a complete format to a floating point format, or vice versa, or from one complete format to another.
- **completeAddition** and **completeSubtraction** add or subtract two complete or floating point format operands, of which at least one is complete, giving a complete format result.
- **completeMultiplyAdd** computes $z + x * y$ where z has a complete format and x, y are of floating point format, giving a complete format result.
- **completeDotProduct**. Let a and b be vectors of length n holding floating point numbers of format \mathbb{F} . Then **completeDotProduct**(a, b) computes $a \cdot b = \sum_{k=1}^n a_k b_k$ exactly and rounds it once to give a result of format \mathbb{F} .

For this final rounding, all 754 rounding modes should be supported. If correct rounding (to nearest) is not provided then faithful rounding (with an error < 1 ulp) shall be provided.

11.12. Recommended operations.

To come shortly.

12. The decoration system at Level 2

12.1. **Decorated interval types.** Formally, the **decorated interval type** \mathbb{DT} derived from a bare interval type \mathbb{T} comprises the set of triples (x, t, d) where (x, t) is a \mathbb{T} -interval tagged with the name t of \mathbb{T} according to §11.4, and $d \in \mathbb{D}$ is a decoration. The members of \mathbb{DT} are **decorated interval datums**. \mathbb{T} is the **parent type** of \mathbb{DT} .


By convention, as with bare intervals (§11.4), the name t is omitted except when clarity requires. Thus \mathbb{T} is regarded as a finite set of mathematical intervals, and a member of \mathbb{DT}

is written as a pair (x, d) , equivalently x_d , where $x \in \mathbb{T}$ and $d \in \mathbb{D}$, that follows the rule for permissible combinations in §10.4.

12.2. Required decorated types. An implementation shall provide the derived decorated interval type for each provided bare interval type. Conversely each provided decorated interval type shall be derived from a provided bare interval type. [Note. That is, the map “is derived from” is a bijection from the set of provided bare interval types to the set of provided decorated interval types, with inverse “is parent of”.]

There shall be a Not an Interval, NaI, datum of each provided decorated interval type. It shall appear to be unique as far as Level 2 operations are concerned, but operations may be provided to set and get a payload in an NaI for diagnostic purposes, in an implementation-defined way (see §10.3).

12.3. Decorated versions of an operation. Let φ be a Level 2 operation whose (input or output) arguments are either of bare interval type, or of a non-interval type (e.g., integer or boolean). A **decorated version** of φ has the same number and order of arguments, with each argument of bare interval type replaced by an argument of the derived decorated interval type; non-interval argument types are unchanged. Further,
[Example. The integer power function $\text{pown}(x, p) = x^p$ has Level 2 bare interval versions with signature $\mathbb{T} = \text{pown}(\mathbb{T}, \mathbb{F})$ where \mathbb{T} is a bare interval type and \mathbb{F} an integer format. Its decorated version has signature $\mathbb{DT} = \text{pown}(\mathbb{DT}, \mathbb{F})$ where \mathbb{DT} is the decorated type derived from \mathbb{T} .]

12.4. Required operations on decorated intervals.  **Elsewhere!** The function `newDec()` shall be provided. Its input is a \mathbb{T} -interval or \mathbb{DT} -interval and its output is a \mathbb{DT} -interval as specified in §10.5.

12.4.1. Interval literals. A decorated interval literal string s is defined to consist of an interval-string sx followed by an underscore followed by a decoration string sd . If sx is a valid interval literal (§11.11.1) representing an interval x according to the implementation, and sd is one of `ill`, `emp`, `trv`, `def`, `dac` or `com` representing the corresponding decoration dx , and if x_{dx} is a permitted combination according to §10.4, then s is **valid** and its value is x_{dx} . In all other cases s is **invalid** and its value is NaI. If the implementation does not support `com`, then `com` is treated as if it were `dac`.

[Note. An implementation may implicitly promote a bare interval literal with value x to the decorated interval `newDec(x)` in suitable contexts, in a language-defined way.]

12.4.2. Interval constants. Each provided Level 2 interval constant (§11.11.2), returning the \mathbb{T} -interval Empty or Entire, shall have a decorated version that returns the \mathbb{DT} -interval `newDec(Empty)` or `newDec(Entire)`, respectively.

12.4.3. Forward-mode elementary functions. Each provided Level 2 arithmetic operation (§11.11.3) with arguments of type \mathbb{T} , shall have a decorated version with corresponding arguments of type \mathbb{DT} . It shall be a decorated interval extension as defined in §10.6—thus the interval part of its output is the same as if the bare interval operation were applied to the interval parts of its inputs.

The only freedom of choice in the decorated version is how the local decoration, denoted dv_0 in (24) of §10.6, is computed. dv_0 shall be the strongest possible (and is thus uniquely defined) if the accuracy mode of the corresponding bare interval operation is “tightest”, but otherwise is only required to obey (24).

12.4.4. Interval case expressions and case function. TBW. Arnold Neumaier had a special recipe, which I need to look up.

12.4.5. Interval-valued non-arithmetic operations. This set of operations includes the reverse-mode elementary functions of §11.11.5, the cancellative addition and subtraction of §11.11.6 and the set operations of §11.11.7. Each provided Level 2 operation of this set, with arguments of type \mathbb{T} , shall have a decorated version with corresponding arguments of type \mathbb{DT} . The decoration shall be as in §10.7.

Versions with alternative decorations such as `intersectionDec` and `convexHullDec`, if provided, should be provided for all supported types.

12.4.6. Constructors. There shall be a decorated version of each provided bare interval constructor of a type \mathbb{T} . It returns a \mathbb{DT} -interval.

`nums2interval.`

For any inputs, if the bare interval constructor succeeds as defined in §11.11.8, and returns a \mathbb{T} -interval \mathbf{x} , the decorated version shall return `newDec(\mathbf{x})`. Otherwise it shall return `NaI`.

text2interval.

If the input string \mathbf{s} is a valid decorated interval literal as defined in §12.4.1, with value \mathbf{x}_{dx} , the constructor shall return \mathbf{x}_{dx} . If \mathbf{s} is a valid bare interval literal as defined in §11.11.1, with value \mathbf{x} , the constructor shall return `newDec(\mathbf{x})`. Otherwise it shall return `NaI`.

12.4.7. *Numeric functions of intervals.* There shall be a decorated version of each provided numeric function of \mathbb{T} -intervals, see §11.11.9. It takes \mathbb{DT} -interval input and the result format is that of the bare interval operation.

Following §10.7, if no input is `NaI`, the result is obtained by discarding the decoration and applying the corresponding bare interval operation. In the case of `NaI` input, the result is `NaN` if the result format supports this, else is language- or implementation-defined.

12.4.8. *Boolean functions of intervals.*

There shall be a decorated version of each provided boolean function of \mathbb{T} -intervals, see §11.11.9. It takes \mathbb{DT} -interval input and the result is boolean.

For the functions in the preceding paragraph, following §10.7, if no input is `NaI` then the result is obtained by discarding the decoration and applying the corresponding bare interval operation. In the case of `NaI` input, the result is `false`, and a language- or implementation-defined exception should be signaled.

There shall be a function `isNaI(\mathbf{X})` with \mathbb{DT} -interval input \mathbf{X} , that returns `true` if \mathbf{X} is `NaI`, else `false`.

12.5. Compressed arithmetic at Level 2. To come shortly.

13. Input and output (I/O) of intervals

13.1. Overview. This clause follows the spirit of the scheme for conversion between floating point numbers and character sequences in the 754 standard, where §5.4.2 and §5.12 specify the mathematical properties of conversion while leaving details, mostly of formatting, language- or implementation-defined. It aims to minimise the risk of incompatible implementations of I/O, while allowing languages and compilers some freedom.

The term *text* denotes character sequences generally, in a language-defined character set, and *string* denotes a particular finite character sequence. The standard is concerned with the conversion between intervals internal to a program, and text. It says nothing about how text may be read from or written to a character stream.

For intervals, containment must hold on both input and output so that, when a program computes an enclosure of some quantity given an enclosure of the data, this remains true all the way from text data to text results.

In addition to normal I/O, the standard requires each interval type \mathbb{T} to have a *public representation*. This has two parts: operations to convert any internal \mathbb{T} -interval x to a string s , and back again to recover x exactly; and documentation of how to convert s to the Level 1 interval represented by x . For proprietary types in particular, this makes explicit the mathematical definition of the type, while letting its Level 3 implementation remain private.

[Note. It follows from the “equal inputs give equal outputs” principle of §11.4 that text output of an interval x , whether by `interval2text` or by `interval2public`, never betrays which of possible alternative internal representations of x was used. E.g., whether a zero bound of an inf-sup interval is stored as -0 or $+0$, or the quantum of a decimal bound (754§2.1.44), shall not be detectable.]

13.2. Input. Implementations shall provide for each supported interval type \mathbb{T} a function

`type-text2interval(s),`

where s is a string. If s is a valid interval literal with value x , the returned value is a \mathbb{T} -interval enclosing x . If s is invalid, Empty is returned.


If \mathbb{T} is a 754-conforming type, the returned value shall be the \mathbb{T} -hull of x . The tightness of enclosure for other types is language- or implementation-defined.

13.3. Output. Implementations shall provide a function

`interval2text(x, cs)`

where x is a bare interval of any supported type \mathbb{T} and cs is a string, the conversion specifier. It converts x to a valid interval literal s whose value encloses x , in a way specified by cs .

The allowed forms of cs are language-defined, and may depend on \mathbb{T} , but shall let the user specify any of the following forms for s , see §11.11.1.

- (i) Inf-sup form $[l, u]$, where the layouts of l and u can be specified independently.
- (ii) Mid-rad form $< m \pm r >$, where the layouts of m and r can be specified independently.
- (iii)  Some of the forms given in the Vienna proposal, Part 6—to be decided.

In either of cases (i) and (ii) the resulting string shall be a valid interval literal.

Here layout of a number means the way it is output as a string. It shall be possible to specify output to a given number of places after the point or to a given number of significant figures. (For instance, by conversion specifiers like `f12.5` and `e12.5` in Fortran, or `%12.5f` and `%12.5e` in C.)

If \mathbb{T} is a 754-conforming type, the enclosure represented by s shall be tightest possible. Namely let $x = [\underline{x}, \bar{x}]$ be a \mathbb{T} -interval. For inf-sup form, l is the largest number of the specified layout that is $\leq \underline{x}$ and u is the smallest number of the specified layout that is $\geq \bar{x}$; either may be infinite in case of overflow. For mid-rad form, m is the number of the specified layout that is closest to the exact midpoint; then r is the smallest number of the specified layout such that the exact interval $[m - r, m + r]$ contains x . The treatment of infinite values, overflow and tie-breaking shall follow that of the `inf`, `sup`, `mid` and `rad` functions in §11.11.9.

For other types the tightness of enclosure of x by s is language- or implementation-defined.

13.4. Public representation. For any supported interval type \mathbb{T} an implementations shall provide functions *type-interval2public* and *type-public2interval*, as follows.

- For any \mathbb{T} -interval datum x the value *type-interval2public*(x) is a string s , the **public representation** of x . It is such that *type-public2interval*(s) is the same as x in the sense of §11.4.

The string s need not contain anything (such as a typename) to identify the type \mathbb{T} .

- The implementation's documentation shall explain how to convert s to the mathematical interval $[l, u]$ represented by the datum x . This shall comprise an effective algorithm for obtaining l and u as decimal or binary numbers, exactly or to any desired accuracy.

A public representation should (this is subjective) be simple. For instance if x represents an interval with small integer bounds such as $[1, 2]$, it should be straightforward to convert s by hand or with the help of a pocket calculator. A good public representation exposes the values of the parameters on which the mathematical model of the type is based.

[*Example. Suppose \mathbb{T} is an inf-sup type whose bounds l, u are rational numbers stored to a fixed number of bits in "floating slash" form. That is, $l = p_l/q_l$ where p_l and q_l ($q_l > 0$) are integers written in binary, occupying k_l and $m - k_l$ bits respectively where k_l (the slash-position) is an integer between 0 and m , and m is a constant of the type. u is similar, involving integers k_u, p_u, q_u . The numbers $k_l, p_l, q_l, k_u, p_u, q_u$ are the parameters of the mathematical model. A good public representation might consist of hexadecimal forms of these six numbers, embedded in suitable punctuation characters.*]

If \mathbb{T} is a 754-conforming type then the public representation s of x shall be a valid interval literal (§11.11.1) that, for nonempty x , is of inf-sup form. Its bounds l, u shall be represented as exact decimal numbers if \mathbb{T} is a decimal type, or in the hexadecimal-significand form of 754§5.12.3 if \mathbb{T} is a binary type.

DRAFT 7.0

Bibliography

- [1] Allen, James F. Maintaining knowledge about temporal intervals. *Communications of the ACM* 26, 832–843, (November 1983).
- [2] Kulisch, Ulrich. Complete Interval Arithmetic and its Implementation on the Computer. Position paper, and the Dagstuhl 2008 proceedings.
- [3] Kulisch, Ulrich. *Computer Arithmetic and Validity: Theory, Implementation, and Applications*. de Gruyter, Berlin, New York, (2008).
- [4] Kulisch, Ulrich and Snyder, Van. *The exact dot product as basic tool for long interval arithmetic*. Position paper, P1788 Working Group, version 11, July 2009.
- [5] Moore, Ramon E. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, N.J., (1966).
- [6] Nehmeier, Marco and Siegel, Stefan and Wolff von Gudenberg, Jürgen. Specification of hardware for interval arithmetic. *Computing* 94, 243-255, (2012).
- [7] Neumaier, Arnold. Vienna Proposal for Interval Standardization. Faculty of Mathematics, University of Vienna, (December 2008). <http://www.mat.univie.ac.at/~neum>
- [8] Pryce, John D. and Corliss, George F. Interval arithmetic with containment sets. *Computing* 78, 251–276, (2006).
- [9] Pryce, John D. P1788 Motion 6: Multi-Format Support: Text and Rationale, (2009).