6. Level 2 description

6.1 Introduction

Entities and operations at Level 2 are said to have **finite precision**. From them, implementable interval algorithms may be constructed. Level 2 entities are called **datums**[4].

6.1.1 Interval type

The interval type, denoted by $\mathbb{T}$, is the **inf-sup type** derived from the IEEE 754 `binary64` format; we refer to the latter as `b64`. This interval type comprises all intervals whose endpoints are `b64` numbers, together with Empty. Since $\pm\infty$ are in `b64`, Entire is in $\mathbb{T}$.

An interval from $\mathbb{T}$ is also called a **bare interval** or a **$\mathbb{T}$-interval**. We use the term **$\mathbb{T}$-datum** to refer to an entity that can be a $\mathbb{T}$-interval or a NaI. A **$\mathbb{T}$-box** is a vector with $\mathbb{T}$-datum components.

6.1.2 Decorated interval type

The decorated interval type, derived from $\mathbb{T}$, is the set of tuples (x, d), where $x \in \mathbb{T}$, and $d \in \mathbb{D}$. We denote this type by $\mathbb{DT}$.

$\mathbb{DT}$ shall contain a "Not an Interval" datum NaI, which is identified with $(\emptyset, \texttt{ill})$.

6.1.3 Operations

The term **$\mathbb{T}$-version** of a Level 1 operation denotes one in which any input or output that is an interval is a $\mathbb{T}$-datum. For bare interval types this includes the following.

– An interval extension (see 6.4) of one of the arithmetic operations of 4.5.

– A set operation, such as intersection and convex hull of $\mathbb{T}$-intervals, returning a $\mathbb{T}$-interval.

– A function such as the midpoint, whose input is a $\mathbb{T}$-interval and output is numeric.

– A constructor, whose input is numeric or text and output is a $\mathbb{T}$-datum.

6.1.4 Exception behavior

For some operations, and some particular inputs, there might not be a valid result. At Level 1 there are several cases when no value exists. However, a Level 2 operation always returns a value. When the Level 1 result does not exist, the corresponding Level 2 operation returns either

– a special value indicating this event (e.g., NaN for most of the numeric functions in 6.7.6); or

– a value considered reasonable in practice. For example, `mid`(Entire) returns 0; a constructor given invalid input returns Empty; and one of the comparisons of 6.7.7, if any input is NaI, returns `false`.

If `intervalPart`() is called with NaI as input, the exception `IntvlPartOfNaI` is signaled (see 6.7.8).

If a bare or decorated constructor fails (see 6.7.5) the exception `UndefinedOperation` is signaled.

If a bare or decorated constructor succeeds on a difficult string argument (see 6.7.5) the exception `PossiblyUndefinedOperation` may be signaled.

6.2 Naming conventions for operations

An operation is generally given a name that suits the context. For example, the addition of two interval datums $\boldsymbol{x}, \boldsymbol{y}$ may be written in generic algebra notation $\boldsymbol{x} + \boldsymbol{y}$; or with a generic text name `add`$(\boldsymbol{x}, \boldsymbol{y})$.

[4] Not "data", whose common meaning could cause confusion.

6.3 Level 2 hull operation

6.3.1 Hull in one dimension

The **interval hull** operation

$$\boldsymbol{y} = \mathrm{hull}(\boldsymbol{s}),$$

maps an arbitrary set of reals $\boldsymbol{s}$ to the tightest interval $\boldsymbol{y}$ enclosing $\boldsymbol{s}$.

6.3.2 Hull in n dimensions

In n dimensions the hull, as defined mathematically in 6.3.1, is extended to act componentwise. That is, for an arbitrary subset $\boldsymbol{s}$ of $\mathbb{R}^n$ it is $\mathrm{hull}(\boldsymbol{s}) = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n)$, where

$$\boldsymbol{y}_i = \mathrm{hull}(\boldsymbol{s}_i),$$

and $\boldsymbol{s}_i = \{\, s_i \mid s \in \boldsymbol{s} \,\}$ is the projection of $\boldsymbol{s}$ on the ith coordinate dimension.

6.4 Level 2 interval extensions

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

$$\boldsymbol{f}(\boldsymbol{x}) \supseteq \mathrm{Rge}(f \,|\, \boldsymbol{x}),$$

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

6.5 Accuracy of operations

This subclause describes requirements and recommendations on the accuracy of operations. Here, *operation* denotes any Level 2 version, provided by the implementation, of a Level 1 operation with interval output and at least one interval input. Bare interval operations are described; the accuracy of a decorated operation is defined to be that of its interval part.

6.5.1 Measures of accuracy

Three **accuracy modes** are defined that indicate the quality of interval enclosure achieved by an operation: *tightest*, *accurate* and *valid* in order from strongest to weakest.

The term **tightness** means the strongest mode that holds uniformly for some set of evaluations. For example, for some one-argument function, an implementation might document the tightness of $\boldsymbol{f}(\boldsymbol{x})$ as being *tightest* for all $\boldsymbol{x}$ contained in $[-10^{15}, 10^{15}]$ and at least *accurate* for all other $\boldsymbol{x}$.

Let $\boldsymbol{f}_{\text{exact}}$ denote the corresponding Level 1 operation. The weakest mode *valid* is just the property of enclosure:

$$\boldsymbol{f}(\boldsymbol{x}) \supseteq \boldsymbol{f}_{\text{exact}}(\boldsymbol{x}). \tag{11}$$

The strongest mode *tightest* is the property that $\boldsymbol{f}(\boldsymbol{x})$ equals $\boldsymbol{f}_{\text{tightest}}(\boldsymbol{x})$, the hull of the Level 1 result:

$$\boldsymbol{f}_{\text{tightest}}(\boldsymbol{x}) = \mathrm{hull}\big(\boldsymbol{f}_{\text{exact}}(\boldsymbol{x})\big). \tag{12}$$

The intermediate mode *accurate* asserts that $\boldsymbol{f}(\boldsymbol{x})$ is *valid*, (11), and is at most slightly wider than the result of applying the *tightest* version to a slightly wider input box:

$$\boldsymbol{f}(\boldsymbol{x}) \subseteq \texttt{nextOut}\left(\boldsymbol{f}_{\text{tightest}}\left(\texttt{nextOut}\big(\mathrm{hull}(\boldsymbol{x})\big)\right)\right). \tag{13}$$

For an interval $\boldsymbol{x}$,

$$\texttt{nextOut}(\boldsymbol{x}) = \begin{cases} [\,\texttt{nextDown}(\underline{x}), \texttt{nextUp}(\overline{x})\,] & \text{if } \boldsymbol{x} = [\underline{x}, \overline{x}] \neq \emptyset, \\ \emptyset & \text{if } \boldsymbol{x} = \emptyset, \end{cases}$$

where `nextUp` and `nextDown` are equivalent to the corresponding functions in IEEE 754.

When $\boldsymbol{x}$ is an interval box, `nextOut` acts componentwise.

NOTE—In (13), the inner `nextOut`() aims to handle the problem of a function such as $\sin x$ 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 `nextOut`() relaxes the requirement for correct (rather than, say, faithful) rounding, which might be hard to achieve for some special functions at some arguments.

6.5.2 Accuracy requirements

Following the categories of functions in Table 4.1, the accuracy of the *basic operations*, the *integer functions* and the *absmax functions* shall be *tightest*. The accuracy of the *cancellative addition and subtraction* operations of 4.5.3 is specified in 6.7.3.

For all other operations in Table 4.1, the accuracy should be *accurate*.

6.5.3 Documentation requirements

An implementation shall document the tightness of each of its interval operations. This shall be done by dividing the set of possible inputs into disjoint subsets ("ranges") and stating a tightness achieved in each range.

[Example. Sample tightness information for the sin function might be

Operation	Tightness	Range
sin	*tightest*	for any $\boldsymbol{x} \subseteq [-10^{15}, 10^{15}]$
	accurate	for all other $\boldsymbol{x}$.

]

Each operation should be identified by a language- or implementation-defined name of the Level 1 operation (which might differ from that used in this standard), its output type, its input type(s) if necessary, and any other information needed to resolve ambiguity.

6.6 Number and interval literals

6.6.1 Number literals

The following forms of number literal shall be provided.

a) A decimal number. This comprises an optional sign, a nonempty sequence of decimal digits optionally containing a point, and an optional exponent field comprising `e` and an integer literal.[5] The value of a decimal number is the value of the sequence of decimal digits with optional point multiplied by ten raised to the power of the value of the integer literal, negated if there is a leading `-` sign.

b) A number in the hexadecimal-floating-constant form of the C99 standard (ISO/IEC9899, N1256 (6.4.4.2)), equivalently hexadecimal-significand form of IEEE Std 754-2008 (5.12.3). This comprises an optional sign, the string `0x`, a nonempty sequence of hexadecimal digits optionally containing a point, and an exponent field comprising `p` and an integer literal exponent. The value of a hexadecimal number is the value of the sequence of hexadecimal digits with optional point multiplied by two raised to the power of the value of the exponent, negated if there is a leading minus sign.

c) A rational literal p `/` q. This comprises an integer literal p, the `/` character, and a positive-natural literal q. Its value is the value of p divided by the value of q.

d) Either of the strings `inf` or `infinity` optionally preceded by `+`, with value $+\infty$; or preceded by `-`, with value $-\infty$.

[5] An integer literal comprises an optional sign and followed by a nonempty sequence of decimal digits.

6.6.2 Bare intervals

The following forms of bare interval literal shall be supported. Below, the number literals l and r are identified with their values. Space shown between elements of a literal denotes zero or more space characters.

– A string `[` l `,` u `]` where l and u are optional number literals of the same radix (10 or 16) with $l \le u$, $l < +\infty$ and $u > -\infty$, see 4.2. Its bare value is the mathematical interval $[l, u]$. Any of l and u may be omitted, with implied values $l = -\infty$ and $u = +\infty$, respectively; e.g. `[,]` denotes Entire.
A string `[` x `]` is equivalent to `[` x `,` x `]`.

– Uncertain form: a string m `?` $r\ v\ E$ where: m is a decimal number literal of form a in 6.6.1, without exponent; r is empty or is a natural-number literal *ulp-count* or is ?; v is empty or is a *direction character*, either `u` (up) or `d` (down); and E is empty or is an *exponent field* comprising the character `e` followed by an integer literal *exponent* e. No whitespace is permitted within the string.

With `ulp` meaning 10^{-d} where d is the number of digits after the decimal point in m (or 0 if there is no decimal point), the literal m`?` by itself denotes m with a symmetrical uncertainty of half an ulp, that is the interval $[m - \frac{1}{2}\mathtt{ulp}, m + \frac{1}{2}\mathtt{ulp}]$. The literal m`?`r denotes m with a symmetrical uncertainty of r ulps, that is $[m - r \times \mathtt{ulp}, m + r \times \mathtt{ulp}]$. Adding `d` (down) or `u` (up) converts this to uncertainty in one direction only, e.g., m`?d` denotes $[m - \frac{1}{2}\mathtt{ulp}, m]$ and m`?`r`u` denotes $[m, m + r \times \mathtt{ulp}]$. Uncertain form with radius `?` is for unbounded intervals, e.g., m`??d` denotes $[-\infty, m]$. The exponent field if present multiplies the whole interval by 10^e, e.g., m `?`r`u` `e`e denotes $10^e \times [m, m + r \times \mathtt{ulp}]$.

– Special values: the strings `[ ]` and `[ empty ]`, whose bare value is Empty, and the string `[ entire ]`, whose bare value is Entire.

6.6.3 Decorated intervals

The following forms of decorated interval literal shall be supported.

– ***sx_sd***: a bare interval literal ***sx***, an underscore "`_`", and a 3-character decoration string ***sd***, where ***sd*** is one of `trv`, `def`, `dac` or `com`, denoting the corresponding decoration dx.

If ***sx*** has the bare value $\boldsymbol{x}$, and if $\boldsymbol{x}_{dx}$ is a permitted combination according to 5.4, then ***sx_sd*** has the value $\boldsymbol{x}_{dx}$. Otherwise ***sx_sd*** has no value as a decorated interval literal.

– The string `[ nai ]`, with the bare value Empty and the decorated value $\text{Empty}_{\mathtt{ill}}$.

The alphanumeric characters in the above literals are case-insensitive (e.g., `[1,1e3]_com` is equivalent to `[1,1E3]_COM`).

6.7 Required operations

Operations in this subclause are described as functions with zero or more input arguments and one return value. It is language-defined whether they are implemented in this way.

6.7.1 Interval constants

There shall be functions `empty`() and `entire`() returning an interval with value Empty and Entire, respectively. There shall also be a decorated version of each, returning

$$\mathtt{newDec}(\text{Empty}) = \text{Empty}_{\mathtt{trv}} \quad \text{and} \quad \mathtt{newDec}(\text{Entire}) = \text{Entire}_{\mathtt{dac}},$$

respectively.

6.7.2 Elementary functions

An implementation shall provide an interval version of each arithmetic operation in Table 4.1. Its inputs and output are intervals, and it shall be a Level 2 interval extension of the corresponding point function. Recommended accuracies are given in 6.5.

NOTE—For operations, some of whose arguments are of integer type, such as integer power $\mathtt{pown}(x, p)$, only the real arguments are replaced by intervals.

Each such operation shall have a decorated version with corresponding arguments of type $\mathbb{DT}$. It shall be a decorated interval extension as defined in 5.6—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 (9) of 5.6, 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 (9).

6.7.3 Cancellative addition and subtraction

An implementation shall provide a $\mathbb{T}$-version of each of the operations `cancelMinus` and `cancelPlus` in 4.5.3. Their inputs and output are $\mathbb{T}$-intervals.

`cancelMinus`$(\boldsymbol{x}, \boldsymbol{y})$ shall return Empty in the first case of (2), the hull of the result in the second, and Entire for each of the cases in (3).

`cancelPlus`$(\boldsymbol{x}, \boldsymbol{y})$ shall be equivalent to `cancelMinus`$(\boldsymbol{x}, -\boldsymbol{y})$.

These operations shall have "trivial" decorated versions, as described in 5.7.

6.7.4 Set operations

An implementation shall provide an interval version of each of the operations `intersection` and `convexHull` in 4.5.4. Its inputs and output are intervals. These operations should return the interval hull of the exact result. If either input to `intersection` is Empty, or both inputs to `convexHull` are Empty, the result shall be Empty.

These operations shall have "trivial" decorated versions, as described in 5.7.

6.7.5 Constructors

For the bare and decorated interval types there shall be a constructor. It returns a $\mathbb{T}$- or $\mathbb{DT}$-datum, respectively.

Bare interval constructors. A bare interval constructor call either **succeeds** or **fails**. This notion is used to determine the value returned by the corresponding decorated interval constructor.

For the constructor `numsToInterval`(l, u), the inputs l and u are `b64` datums. If neither l nor u is NaN, and $l \le u$, $l < +\infty$, $u > -\infty$, the result is `[`l`,`u`]`. Otherwise the call fails, and the result is Empty.

For the constructor `textToInterval`$(\boldsymbol{s})$, the input $\boldsymbol{s}$ is a string. The string of form `[` l `,` u `]` where $l < +\infty$ and $u > -\infty$ are optional number literals is called **difficult** if either one of the literals is a rational number literal of form c) in 6.6.1 or one of them is a decimal number literal of form a) in 6.6.1 and another is a hexadecimal number literal of form b) in 6.6.1.

– If $\boldsymbol{s}$ is a valid interval literal with Level 1 value $\boldsymbol{x}$ and $\boldsymbol{s}$ is not difficult, the result shall be the hull of $\boldsymbol{x}$ (the constructor succeeds).

– If $\boldsymbol{s}$ is not a valid interval interval and $\boldsymbol{s}$ is not difficult, this constructor fails, and the result is Empty.

– If $\boldsymbol{s}$ is difficult and $l \le u$, the result may be any interval containing `[`l`,`u`]` (the constructor succeeds).

– If $\boldsymbol{s}$ is difficult and $l > u$, the result may be either any interval containing `[`u`,`l`]` (the constructor succeeds), or Empty (the constructor fails).

Decorated interval constructors. Let the prefix `b-` or `d-` denote the bare or decorated version of a constructor. If `b-numsToInterval`(l, u) or `b-textToInterval`$(\boldsymbol{s})$ succeeds with result $\boldsymbol{y}$, then `d-numsToInterval`(l, u) or `d-textToInterval`$(\boldsymbol{s})$, respectively, succeeds with result $\boldsymbol{y}$ and decoration `newDec`$(\boldsymbol{y})$, see 5.5.

If $\boldsymbol{s}$ is a decorated interval literal $\boldsymbol{sx}\_\boldsymbol{sd}$ with Level 1 value $\boldsymbol{x}_{dx}$, see 6.6.3, and `b-textToInterval`$(\boldsymbol{sx})$ succeeds with result $\boldsymbol{y}$, then `d-textToInterval`$(\boldsymbol{s})$ succeeds with result $\boldsymbol{y}_{dy}$, where $dy = dx$ except when $dx =$ `com` and overflow has occurred, that is, $\boldsymbol{x}$ is bounded and $\boldsymbol{y}$ is unbounded. Then dy shall equal `dac`.

Otherwise the call fails, and the result is NaI.

Exception behavior. Exception `UndefinedOperation` is signaled by both the bare and the decorated constructor when the input is such that the bare constructor fails.

Exception `PossiblyUndefinedOperation` is signaled by both the bare and the decorated `textToInterval`($\boldsymbol{s}$) constructor with difficult $\boldsymbol{s}$

– when $l > u$ and interval constructor succeeds;

– optionally when $l \leq u$.

NOTE—When signaled by the decorated constructor it will normally be ignored since returning NaI gives sufficient information.

NOTE—The behavior of the `textToInterval`($\boldsymbol{s}$) constructor is implementation-dependent when $\boldsymbol{s}$ is difficult. The least accurate implementation simply returns Entire and signals `PossiblyUndefinedOperation`. The most accurate implementation fails with `UndefinedOperation` exception when $l > u$ and returns the hull of `[`l`,`u`]` without exception otherwise.

6.7.6 Numeric functions of intervals

An implementation shall provide a $\mathbb{T}$-version of each numeric function in Table 4.3 of 4.5.6 giving a result in `b64`. The mapping of a Level 1 value to a `b64` number is defined in terms of the following rounding methods:

Round toward positive: x maps to the smallest `b64` number not less than x; 0 maps to +0.

Round toward negative: x maps to the largest `b64` number not greater than x; 0 maps to −0.

Round to nearest: x maps to the `b64` number (possibly $\pm\infty$) closest to x as defined by IEEE 754-2008; 0 maps to +0.

NOTE—These functions help define operations of the standard but are not themselves operations of the standard.

A Level 1 value of 0 shall be returned as −0 by `inf`, and +0 by all other functions in this subclause.

`inf`($\boldsymbol{x}$) returns the Level 1 value rounded toward negative.

`sup`($\boldsymbol{x}$) returns the Level 1 value rounded toward positive.

`mid`($\boldsymbol{x}$): the result is defined by the following cases, where $\underline{x}, \overline{x}$ are the exact (Level 1) lower and upper bounds of $\boldsymbol{x}$:

$\boldsymbol{x}$ = Empty	NaN
$\boldsymbol{x}$ = Entire	0
$\underline{x} = -\infty$, $\overline{x}$ finite	the finite negative `b64` number of largest magnitude
$\underline{x}$ finite, $\overline{x} = +\infty$	the finite positive `b64` number of largest magnitude
$\underline{x}$, $\overline{x}$ both finite	the Level 1 value rounded to nearest

The implementation shall document how it handles the last case.

`rad`($\boldsymbol{x}$) returns NaN if $\boldsymbol{x}$ is empty, and otherwise the smallest `b64` number r such that $\boldsymbol{x}$ is contained in the exact interval $[m - r, m + r]$, where m is the value returned by `mid`($\boldsymbol{x}$).

`wid`($\boldsymbol{x}$) returns NaN if $\boldsymbol{x}$ is empty. Otherwise it returns the Level 1 value rounded toward positive.

`mag`($\boldsymbol{x}$) returns NaN if $\boldsymbol{x}$ is empty. Otherwise it returns the Level 1 value rounded toward positive.

`mig`($\boldsymbol{x}$) returns NaN if $\boldsymbol{x}$ is empty. Otherwise it returns the Level 1 value rounded toward negative, except that 0 maps to +0.

Each bare interval operation in this subclause shall have a decorated version, where each input of bare interval type is replaced by one of the corresponding decorated interval type, and the result format is that of the bare operation. Following 5.7, if any input is NaI, the result is NaN. Otherwise the result is obtained by discarding the decoration and applying the corresponding bare interval operation.

6.7.7 Boolean functions of intervals

An implementation shall provide the functions

– `isEmpty`($\boldsymbol{x}$) and `isEntire`($\boldsymbol{x}$) in 4.5.7,

– the functions implementing the comparison relations in Table 4.5 of 4.5.7, and

– the function `isNaI`($\boldsymbol{x}$) for input $\boldsymbol{x}$ of any decorated type, which returns `true` if $\boldsymbol{x}$ is NaI, else `false`.

Each bare interval operation in this subclause shall have a decorated version. Following 5.7, if any input is NaI, the result is `false` (in particular `equal`(NaI, NaI) is `false`). Otherwise the result is obtained by discarding the decoration and applying the corresponding bare interval operation.

6.7.8 Operations on/with decorations

An implementation shall provide the operations of 5.5. These comprise the comparison operations $=$, $\neq$, $>$, $<$, $\geq$, $\leq$ for decorations; and, for the decorated type, the operations `newDec`, `intervalPart`, `decorationPart` and `setDec`.

A call `intervalPart`(NaI), whose value is undefined at Level 1, shall return Empty at Level 2, and shall signal the `IntvlPartOfNaI` exception to indicate that a valid interval has been created from the ill-formed interval.