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.

Normative references. TBW .(Subsection not numbered for now, not to upset existing numbering during discussion.)

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.
- 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 program data types, or bit patterns, except for interval types derived from IEEE 754 floating point formats.

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 7 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. Ideas underlying the standard (informative)

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

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

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 [**4**] uses only bounded, closed, nonempty intervals in the real numbers $\mathbb{R}$—that is, $[a, b] = \{\, x \in \mathbb{R} \mid a \le x \le b \,\}$ where $a, b \in \mathbb{R}$ with $a \le b$. 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 [**7**], 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 $x \to +\infty$, 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 $a \le b$ is "proper" and identified with the normal interval $\{\, x \in \mathbb{R} \mid a \le x \le b \,\}$, 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.

2.2. Specification Levels. The 754-2008 standard describes itself as layered into four Specification Levels. To manage complexity, P1788 uses a corresponding structure. It deals mainly with Level 1, of mathematical *interval theory*, and Level 2, the finite set of *interval datums* in terms of which finite-precision interval computation is defined. It has some concern with Level 3, of *representations* of intervals as data structures; and none with Level 4, of *bit strings* and memory.

There is another important player: the programming language. 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.

2.3. The Fundamental Theorem. Moore's [**4**] Fundamental Theorem of Interval Arithmetic (FTIA) is central to interval computation. Roughly, it says as follows. Let f be an *explicit arithmetic expression*—that is, it is built from finitely many elementary functions (arithmetic operations) such as $+, -, \times, \div, \sin, \exp, \ldots$, with no non-arithmetic operations such as intersection, so that it defines a real function $f(x_1, \ldots, x_n)$. Then evaluating f "in interval mode" over any interval inputs $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ 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.

2.4. Operations.

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

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

(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 $\boldsymbol{y} = [0, 2]$, 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.

2.5. Decorations.

Many interval algorithms are only valid if certain mathematical conditions are satisfied: for instance one may need to know that a function, defined by an expression, is everywhere continuous on a box in $\mathbb{R}^n$ defined by n input intervals $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. 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 §2.4. 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 a *compressed* decorated interval scheme is provided for advanced use. It aims to give the speed of 16-byte objects, at a cost in flexibility but supporting applications such as checking whether a function is defined and continuous on its inputs.