The MPFI Library: Towards IEEE 1788–2015 Compliance

(*In Memoriam Dmitry Nadezhin*)

Nathalie Revol(✉)

University of Lyon - Inria, LIP - ENS de Lyon, 46 allée d'Italie, 69007 Lyon, France
Nathalie.Revol@inria.fr
http://perso.ens-lyon.fr/nathalie.revol/

Abstract. The IEEE 1788–2015 has standardized interval arithmetic. However, few libraries for interval arithmetic are compliant with this standard. In the first part of this paper, the main features of the IEEE 1788–2015 standard are detailed, namely the structure into 4 levels, the possibility to accomodate a new mathematical theory of interval arithmetic through the notion of *flavor*, and the mechanism of *decoration* for handling exceptions. These features were not present in the libraries developed prior to the elaboration of the standard. MPFI is such a library: it is a C library, based on MPFR, for arbitrary precision interval arithmetic. MPFI is not (yet) compliant with the IEEE 1788–2015 standard for interval arithmetic: the planned modifications are presented. Some considerations about performance and HPC on interval computations based on this standard, or on MPFI, conclude the paper.

Keywords: Interval arithmetic · IEEE 1788–2015 standard · MPFI library · Compliance

1 Introduction and Context

Interval arithmetic has been defined even before the 1960s [15,27] and has continuously evolved and improved since then, with the development of algorithms to solve larger classes of problems through the 1970s and 1980s [1,16,20], then with a focus on the implementation [26] and more recently with its introduction in master courses [17,29]. However, in 2008, it was noticed and strongly resented that there were no definitions common to all authors and that it made it difficult to compare results. Under the auspices of IEEE where the standardization of floating-point arithmetic took place, leading to IEEE 754–1985 [8] and IEEE 754–2008 [9], a standardization effort was launched. It led to the IEEE 1788–2015 standard for interval arithmetic [10]. Its development phases and its main features are explained in [14,21,25].

Nevertheless, only few libraries of interval arithmetic are compliant with the IEEE 1788–2015 standard. Most libraries were developed before the standard. Regarding the libraries developed since then and compliant with the standard,

R. Wyrzykowski et al. (Eds.): PPAM 2019, LNCS 12044, pp. 353–363, 2020.
https://doi.org/10.1007/978-3-030-43222-5_31

let us mention `libieee1788` [19], which was developed along with the standard as a proof-of-concept. However, its author has left academia and this C++ library is no longer maintained: it does not compile any more with recent versions of g++. `JInterval` [18] is a Java library that was more used to test the compliance of interval arithmetic libraries with the standard, than used as an interval arithmetic library *per se*. Unfortunately, this library has also untimely lost its main developer, D. Nadezhin. The `interval` package for Octave [7] is the only library that is still maintained and for public use, even if its author has also left academia. `MPDI` [5] is another library for interval arithmetic that is compliant with the standard, but it is not (yet?) distributed.

This lack of libraries compliant with the IEEE 1788–2015 standard led us to consider the adaptation of our MPFI library for arbitrary precision interval arithmetic into a compliant library. We will detail the required modifications in Sect. 4, but we first detail the main features of the IEEE 1788–2015 standard in Sect. 2, and we introduce MPFI and explain how far it is from being IEEE 1788–2015 compliant in Sect. 3. We conclude with some personal considerations about the relevance of interval arithmetic computations in HPC.

2 IEEE 1788–2015 Standard for Interval Arithmetic

2.1 Structure in Four Levels

The IEEE 1788–2015 standard for interval arithmetic is structured in 4 levels, similarly to the IEEE 754–1985 standard for floating-point arithmetic. This structure clearly separates the mathematical notions from implementation issues, all the way to bit encoding.

The first level is the mathematical level: this level is about intervals of real numbers, such as $[0, \pi]$. Reasoning about intervals of real numbers, establishing and proving mathematical theorems about such intervals, are done at Level 1. No representation issue interferes with this level.

The second level addresses discretization issues: it deals with the fact that an implementation on a computer will have a discrete, finite set of numbers at its disposal for the representation of intervals, in particular for the representation of the endpoints. It specifies the existence of directed rounding modes, because it is required that every interval at Level 2 encloses the mathematical, Level 1, interval it represents. For instance, the interval $[0, \pi]$ is represented by an interval of the form $[\mathrm{rd}(0), \mathrm{ru}(\pi)]$ where rd stands for *rounding downwards* and ru stands for *rounding upwards*. This second level remains quite abstract and does not specify the set of numbers, it – only but completely – explicitly specifies how to go from the real numbers at mathematical Level 1 to a finite and discrete set of numbers at Level 2.

At Level 3, this finite and discrete set of numbers is specified: for instance it can be the set of `binary64` floating-point numbers given by the IEEE 754–2008 standard. Level 4 gives the binary encoding of the representation. Actually, the bulk of the work has been done at the floating-point (or any other numbers representation) level and the standard specifies only decorations, see Sect. 2.4.

2.2 Definitions: Intervals and Operations

Notation: following [13], intervals are denoted using boldface symbols, as in $\mathbf{x}$.

Now that the structure of the standard is clear, let us detail the definitions adopted in the standard. Regarding intervals: everybody agreed that $[0, \pi]$ and $[1, 3]$ are intervals. Discussions were hot regarding whether $\emptyset$, $[5, \infty)$ or $[3, 1]$ should be considered as legal intervals. Thus, at Level 1, the definition for which there was a consensus, a **common** agreement, is that an interval is a non-empty bounded closed connected subset of $\mathbb{R}$: $\mathbf{x} = [\underline{x}, \bar{x}]$ with $\underline{x} \in \mathbb{R}$, $\bar{x} \in \mathbb{R}$ and $\underline{x} \leq \bar{x}$.

At Level 1, operations are defined in such a way that the FTIA holds.

Theorem 1 (FTIA: Fundamental Theorem of Interval Arithmetic). *Any operation φ evaluated on interval arguments $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k$ within its domain returns its range on these arguments $\varphi(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k)$.*

Implementation issues relax the FTIA to the requirement that the result encloses the range of φ on $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k$. The application of this principle yields the well-known formulas for arithmetic operations such as $+$, $-$, $*$ or $\sqrt{}$:

$$
\begin{aligned}
&[\underline{x}, \bar{x}] + [\underline{y}, \bar{y}] = [\underline{x} + \underline{y}, \bar{x} + \bar{y}],\\
&[\underline{x}, \bar{x}] - [\underline{y}, \bar{y}] = [\underline{x} - \bar{y}, \bar{x} - \underline{y}],\\
&[\underline{x}, \bar{x}] * [\underline{y}, \bar{y}] = [\min(\underline{x} * \underline{y}, \underline{x} * \bar{y}, \bar{x} * \underline{y}, \bar{x} * \bar{y}), \max(\underline{x} * \underline{y}, \underline{x} * \bar{y}, \bar{x} * \underline{y}, \bar{x} * \bar{y})],\\
&\sqrt{[\underline{x}, \bar{x}]} = [\sqrt{\underline{x}}, \sqrt{\bar{x}}] \text{ if } \underline{x} \geq 0,
\end{aligned}
$$

and is used to evaluate mathematical functions, e.g. $\sin([3, 5]) \subset [-1, +0.14113]$.

Other operations are also specified by the IEEE 1788–2015 standard. Some operations are specific to sets, such as the intersection or the convex hull of the union, for instance $[2, 4] \cap [3, 7] = [3, 4]$ and $[-2, -1] \cup [3, 7] = [-2, 7]$. In the latter example, the closed convex hull of the result of the union must be returned, otherwise the result has a "gap" and is not an interval. Some operations are specific to intervals, such as the endpoints (infimum and supremum): $\inf([-1, 3]) = -1$, $\sup([-1, 3]) = 3$; the width and the radius: $\mathrm{wid}([-1, 3]) = 4$, $\mathrm{rad}([-1, 3]) = 2$; the magnitude and the mignitude[1]: $\mathrm{mag}([-1, 3]) = 3$, $\mathrm{mig}([-1, 3]) = 0$.

Some operations have been added to ease constraint solving: it is known that the addition and subtraction defined above are not the reciprocal of each other. The standard specifies two operations that are respectively the reciprocal of addition, namely `cancelMinus`, and of subtraction, namely `cancelPlus`. The formulas for `cancelMinus` and `cancelPlus` are as follows

$$
\begin{aligned}
&\texttt{cancelMinus}(\mathbf{x}, \mathbf{y}) && = \mathbf{z} \text{ such that } \mathbf{y} + \mathbf{z} = \mathbf{x} && \\
\Rightarrow\ &\texttt{cancelMinus}([\underline{x}, \bar{x}], [\underline{y}, \bar{y}]) && = [\underline{x} - \underline{y}, \bar{x} - \bar{y}], && \text{if } \mathrm{wid}(\mathbf{x}) \geq \mathrm{wid}(\mathbf{y}),\\
&\texttt{cancelPlus}(\mathbf{x}, \mathbf{y}) && = \texttt{cancelMinus}(x, -y) && = \mathbf{z} \text{ such that } \mathbf{z} - \mathbf{y} = \mathbf{x},\\
\Rightarrow\ &\texttt{cancelPlus}([\underline{x}, \bar{x}], [\underline{y}, \bar{y}]) && = [\underline{x} + \bar{y}, \bar{x} + \underline{y}], && \text{if } \mathrm{wid}(\mathbf{x}) \geq \mathrm{wid}(\mathbf{y}).
\end{aligned}
$$

For example, $\texttt{cancelMinus}([2, 5], [1, 3]) = [1, 2]$ and $\texttt{cancelPlus}([2, 5], [1, 3]) = [5, 6]$. Such reciprocal operations are called *reverse operations*.

[1] The definition of the mignitude is $\mathrm{mig}([a, b]) = \min(|x| : x \in [a, b]) = \min(|a|, |b|)$ if $0 \notin [a, b]$ and 0 otherwise.

2.3 Flavors

This definition of an interval and the specification of these operations are the common ground of the IEEE 1788–2015 standard for interval arithmetic. However, this common ground was felt as too restrictive by many users of interval arithmetic, who are accustomed to manipulate a larger set of intervals in their daily practice. Still, it was impossible to extend the definition of an interval to simultaneously encompass all varieties of intervals and still keep a consistent theory. For instance, both $\emptyset$ and $[5, +\infty)$ are meaningful within the set-based approach of interval arithmetic, but not $[3, 1]$. Conversely, $[3, 1]$ is a valid interval in Kaucher [12] or modal arithmetic, but neither $\emptyset$ nor unbounded intervals.

The standard is thus designed to accomodate "variants" of interval arithmetic, called *flavors* in IEEE 1788–2015. After many discussions, including a clear definition of modal arithmetic [3,4], the partisans of modal arithmetic did not pursue their standardization effort. Currently, only the *set-based flavor*, derived from set theory, is specified by the IEEE 1788–2015 standard.

Let us highlight the set-based flavor. First, the set-based flavor removes some limitations on the allowed intervals: the empty set as well as unbounded intervals are legal intervals for this flavor. An interval is a closed connected subset of $\mathbb{R}$. As the empty set is a valid interval, the definition of operations and functions can be extended outside their domain, and $\sqrt{[-1, 2]}$ now has a meaning. More generally, the meaning of $\varphi(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k)$ is

$$\varphi(\mathbf{x}_1, \ldots, \mathbf{x}_k) = \text{Hull}\left\{\varphi(x_1, \ldots, x_k) \ : \ (x_1, \ldots, x_k) \in (\mathbf{x}_1, \ldots, \mathbf{x}_k) \cap \text{Dom}(\varphi)\right\}.$$

Let us go back to the example given above: $\sqrt{[-1, 2]} = \sqrt{[-1, 2] \cap Dom_{\sqrt{}}} = \sqrt{[0, 2]} = [0, \sqrt{2}]$. Similarly, $[2, 3]/[-1, 2]$ is permitted and $[2, 3]/[-1, 2] = \mathbb{R}$, whereas $[2, 3]/[0, 2] = \text{Hull}([2, 3]/(0, 2]) = [1, +\infty)$.

Another extension defined by the set-based flavor is the set of available operations, in particular of reverse operations. For instance, the reverse operation of the square operation is `sqrRev`, examplified here:

$$\texttt{sqrRev}([1, 4]) = \text{Hull}\left\{x \ : \ x^2 \in [1, 4]\right\} = \text{Hull}\left([-2, -1] \cup [1, 2]\right) = [-2, 2].$$

Another important reverse operation is `mulRevToPair`, that corresponds to the extended division defined in [22]. This operation is rather peculiar, as it returns 0, 1 or 2 interval(s), as in $\texttt{mulRevToPair}([2, 3], [-1, 2]) = ((-\infty, -2], [1, +\infty))$. It does not return the convex hull of the result, instead it preserves the gap. This is particularly useful in Newton's method for the determination of the zeroes of a function: as this gap corresponds to a region that does not contain any zero and that can be eliminated for further exploration, it also separates zeros. Newton's method can subsequently be applied successfully to each of the two results.

2.4 Decorations

Let us have a closer look at Newton's method. A particularly useful side result is the proof of existence, and sometimes uniqueness, of a zero in the computed interval. This proof is obtained by applying Brouwer's theorem.

Theorem 2 (Brouwer's Theorem). *If the image of a compact set K by a continuous function f is enclosed in K, then f has a fixed point in K: if $f(K) \subset K$, then $\exists x_0 \in K$ such that $f(x_0) = x_0$.*

Another way of stating this result is to say that the function $g : x \mapsto x - f(x)$ has a zero in K.

In particular, if K is a non-empty bounded interval, and if the result of the evaluation of f on K returns an interval $K' \subset K$, then the existence of a fixed-point of f on K is established.

Let us consider the following example: the function

$$f \ : \ x \mapsto \sqrt{x} - 2,$$

has no real fixed point. We leave it to the reader, hint: $x - \sqrt{x} + 2$ has no real zero, or equivalently the polynomial $y^2 - y + 2$ has no real root. The evaluation of f on $[-4, 9]$ using the set-based flavor of interval arithmetic yields:

$$\sqrt{[-4, 9]} - 2 = \sqrt{[0, 9]} - 2 = [0, 3] - 2 = [-2, 1] \subset [-4, 9],$$

and a hasty application of Brouwer's theorem falsely establishes that f has a fixed point in $[-4, 9]$. The mistake here is to omit checking whether f is continuous over $[-4, 9]$. Actually f is not even defined everywhere on $[-4, 9]$. As the assumption of Brouwer's theorem is not satisfied, no conclusion can be derived.

The IEEE 1788–2015 standard must offer a mechanism to handle exceptions and to prohibit such erroneous conclusions from being drawn. After hot and long debates, the chosen mechanism is called *decoration*; it consists in a piece of information, a tag attached to or "decorating" each interval. Decorations have been deemed as the best way (or, should we say, "the least worse") to deal with the abovementioned problem:

- they avoid the inappropriate application of Brouwer's theorem: Brouwer's theorem can be used only when the tag indicates that it is valid to do so;
- they avoid the storage of any global information for exceptions, contrary to the global flags defined in the IEEE 754–1985 standard for floating-point arithmetic: such global flags are difficult to implement in a parallel context (that is, SIMD, multithreaded, or distributed).

The meaning of a decoration, in the set-based flavor, is a piece of information about the history of the computations that led to the considered interval. In particular, it indicates whether every operation involved a defined and continuous function applied to arguments within its domain or not. The user must thus consult this decoration before applying Brouwer's theorem for instance.

For the set-based flavor, the chosen decorations are listed below:

- `com` for common,
- `dac` for defined and continuous,
- `def` for defined,
- `trv` for trivial (no information),
- `ill` for ill-formed (nowhere defined).

As a decoration results from the computation of the interval it is attached to, this computation must also incorporate the determination of the decoration. The set-based flavor specifies the propagation rules for decorations.

Last, every flavor must provide a FTDIA (Fundamental Theorem of Decorated Interval Arithmetic), that accounts for decorations.

Theorem 3 (FTDIA for the Set-Based Flavor). *Let* `f` *be an arithmetic expression denoting a real function f. Let* `f` *be evaluated, possibly in finite precision, on a validly initialized decorated box $\mathbf{X} = \mathbf{x}_{dx}$, to give result $\mathbf{Y} = \mathbf{y}_{dy}$. If some component of $\mathbf{X}$ is decorated* `ill`*, then the decoration $dy =$* `ill`*. If no component of $\mathbf{X}$ is decorated* `ill`*, and none of the operations φ of f is an everywhere undefined function, then $dy \neq$* `ill` *and $\mathbf{y} \supset \mathrm{Range} f(\mathbf{x})$ and the decoration dy of $\mathbf{y}$ correctly (i.e., pessimistically) accounts for the properties of f over $\mathbf{x}$.*

By pessimistically, it is expected that a decoration never raises false hopes. For instance, a function can be defined and continuous while the computed decoration only states `def`, but the converse cannot happen.

2.5 Exact Dot Product

As the IEEE 1788–2015 standard addresses the quality of numerical computations, it also incorporates a recommendation regarding the accuracy of specific floating-point computations. Namely, it recommends that for each supported IEEE 754–1985 floating-point type, an implementation shall provide a correctly rounded version of the four reduction operations `sum`, `dot`, `sumSquare` and `sumAbs`, that take a variable number of arguments.

3 The MPFI Library

After this introduction to the IEEE 1788–2015 standard for interval arithmetic, let us now concentrate on its implementation. As already stated, the libraries that are compliant with the standard are rather rare. This section focuses on the MPFI library, developed since 2000 and thus prior to the standard by large, and on its transformation into an IEEE 1788–2015 compliant library.

MPFI stands for *Multiple Precision Floating-point reliable Interval* library. It is a library written in C that implements arbitrary (rather than multiple) precision interval arithmetic. More precisely, intervals are represented by their endpoints and these endpoints are floating-point numbers of arbitrary precision: for each endpoint, the significand can be arbitrarily precise, the only limit being the memory of the computer. The MPFI library is based on MPFR [2] for arbitrary precision floating-point arithmetic. Its development started in 2000 with

Revol and Rouillier [24], it has evolved and improved since then thanks to the contributions of S. Chevillard, C. Lauter, H. D. Nguyen and Ph. Théveny. The library is freely available at https://gforge.inria.fr/projects/mpfi/.

Before digging in the functionalities and specificities of MPFI, let us recall some justification for its development, as given by Kahan in [11]. In "How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?", Kahan lists tools for assessing the numerical quality of a computed result, in the presence of roundoff errors. He exhibits examples that defeat these tools, when applied mindlessly. A typical example of mindless use of a tool such as interval arithmetic is the replacement of every floating-point datatype in the code by an interval datatype that is not more precise, before running the code again, on data of interval type(s). It is well known that, most of the time, such a mindless use of interval arithmetic produces results with widths too large to be of any help. However, if running time is not an issue, using interval arithmetic with arbitrary precision, and increasing the precision as needed, is a mindless (as opposed to artful, or expert) but cheap (in development time) and effective way to assess the numerical quality of a code. As Kahan puts it [11], "*For that price* (slow execution compared to the execution of the purely floating-point version) *we may be served better by almost foolproof extendable-precision Interval Arithmetic.*". MPFI offers the required arbitrary precision interval arithmetic.

Let us go back to MPFI and detail the definitions it uses and the functionalities it offers. MPFI is based on MPFR and thus on GMP, for accuracy, efficiency and portability. As GMP and MPFR, MPFI is developed in the C language. MPFR provides arbitrary precision floating-point arithmetic, that is compliant with the IEEE 754–1985 philosophy. In particular, for every function, MPFR guarantees that the returned result is equal to the exact result (that is, as if it were computed with infinite precision), rounded using the required rounding mode. This correct rounding is guaranteed not only for basic arithmetic operations but for every function of the mathematical library. In MPFI, an interval is any closed connected subset of $\mathbb{R}$ whose endpoints are numbers representable using MPFR. Thus the empty set and unbounded intervals are valid intervals, However, this definition corresponds to Level 2 of the IEEE 1788–2015 standard.

Regarding the functionalities offered by MPFI, they correspond to most of the requirements of IEEE 1788–2015, with some exceptions. On the one hand, MPFI offers a richer set of mathematical constants (π, Euler constant, etc.) and functions. On the other hand, there is (yet) no implementation of the reverse functions, except `mulRevToPair`. MPFI offers most of the lengthy list of conversions mandated by the standard: to and from integer, double precision floating-point numbers, exact integers and rationals (through GMP), arbitrary precision floating-point numbers (through MPFR) and text strings. MPFI also accomodates intervals with any floating-point endpoints, including infinities, signed zeroes and NaNs: again, MPFI has been designed at Level 2 of IEEE 1788–2015.

However, MPFI accounts for neither flavors nor decorations. Thus, operations are not defined according to any flavor and do not propagate decorations. Still, MPFI has a mechanism for handling exceptions, which is a "Level 2"

mechanism in the sense that it is based on the floating-point, IEEE 754–1985-like, mechanism for handling exceptions. Let us illustrate this mechanism through an example: when MPFI is given $\sqrt{[-1,2]}$, as $[-1,2]$ contains -1 and as $\sqrt{-1}$ is an invalid result denoted as `NaN` in floating-point arithmetic, MPFI considers this computation as an invalid one and returns `NaI`: Not an Interval. In IEEE 1788–2015, the only NaIs are produced by meaningless inputs such as ["bla", 1].

To sum up, MPFI has to be reworked in several directions to be compliant with the IEEE 1788–2015 standard.

4 Towards Compliance of MPFI with IEEE 1788–2015

In order to incorporate the new concepts present in the standard, the *data structure* of a MPFI interval must be modified. First, a field `flavor` will be added to each interval, even if this was not the original intent of IEEE 1788–2015: the principle of flavors was that either a whole computation, or at least significant portions of it, would be performed using a single flavor; thus a flavor would be attached to a computation rather than to a data.

Second, parameterized by the flavor, a field `decoration` will be added and its possible values will be the ones defined by the corresponding flavor. The technicalities of "bare" intervals and "compressed" intervals will be handled in an ad hoc way (by adding a boolean indicating whether the interval is bare or not) or not implemented (in the case of compressed intervals). As these notions were not detailed in Sect. 2, they will not be discussed further here.

Then, the *code for each existing operation* needs to be updated. When entering the code of an operation, a preliminary test on the flavors of the arguments and on their compatibility will be performed, and the computation will then be branched to the corresponding part of the code. Before quitting the code, a postprocessing will be performed to determine and set the decoration of the result. Code for the missing reverse operations must be developed.

Another issue is *backward compatibility* for users of MPFI who want to preserve the existing behavior of their MPFI computations. This will be achieved by adding a new "flavor" – even if it is not really one: no clear specification at Level 1 – called `MPFIoriginal`, so that every computation behaves the same old way. When this flavor is encountered, each operation will branch to the existing and unmodified code to perform it.

5 Concluding Remarks Regarding Performance and HPC

The previous section is written in future tense, because most of the modifications are still waiting to be implemented. Indeed, a major update of MPFI is ongoing, but still not finished. This update consists not only in turning MPFI into a IEEE 1788–2015 compliant library, but also in incorporating all mathematical functions provided by MPFR, such as erf or Bernoulli. Another direction of future developments concerns the ease of use of MPFI, through a Julia interface.

Let us now conclude with a few remarks regarding performance and HPC. The author worked on the parallelization [23] of Hansen's algorithm for global optimization using interval arithmetic [6]. This algorithm is of branch-and-bound type and the original idea to parallelize it was to explore simultaneously several branches of the tree corresponding to the branch-and-bound exploration. However, it was rapidly obvious that brute force (that is, bisection of the candidate box and evaluation of the objective function over each sub-box) was not the best way to obtain speed-ups. A smarter, sequential processing of the candidate box was more efficient, either to reduce it or to prune it. The simplest solution was, as mentioned in [11], to use larger or arbitrary precision interval arithmetic. This led to the development of MPFI.

Let us go back to parallel computations, with "parallel" covering a large spectrum of possibilities, all the way from SIMD to multithreaded to multicore to distributed to heterogeneous computations. The IEEE 1788–2015 standard has tried to avoid some pitfalls, such as the use of global flags for handling exceptions. However, the mechanism of decorations has also been heavily criticized. On the one hand, adding this piece of information to each interval destroys padding efforts and other memory optimizations. On the other hand, the computation and propagation of decorations does not integrate gracefully with pipelined or SIMD operations such as AVX, SSE or SSE2. Similarly, MPFI computations do no seem suited for parallel execution. The MPFI library cannot benefit from hardware accelerators. It is also not well suited to cache optimizations strategies, as its data have irregular sizes, as opposed to fixed and constant sizes such as `binary32` or `binary64` floating-point datatypes. Furthermore, each operation in MPFI takes a large computing time, compared to the time of the same operation (such as addition, multiplication or exponential) applied to `binary64` operands. In practice, a slowdown larger than 50, for one operation, has often been observed.

However, IEEE 1788–2015 and MPFI computations are not comparable with `binary32` or `binary64` computations. First, the results they provide are guaranteed, in the sense that they contain the sought results, even in the presence of roundoff errors. Second, they are well suited for multithreaded or distributed computations: for such computations, it is well known that the communication time needed to bring the data to the computational device is much larger, by typically 3 orders of magnitude, than the computational time, that is, the time required to perform the arithmetic operations on these data. It means that there is plenty of time to apply numerical computations to the data. With interval computations and, in particular, with arbitrary precision interval computations, the computational time is much larger and becomes closer to the communication time. In other words, with interval computations, the numeric intensity is increased, as already observed in [28, Chapter 8] for the product of interval matrices with `binary64` coefficients. HPC computations leave time for interval computations and high-precision interval computations: the execution time is better balanced between communication time and computation time, with a better final accuracy and a guarantee on the results.

References

1. Alefeld, G., Herzberger, J.: Introduction to Interval Analysis. Academic Press, Cambridge (1983)
2. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: a multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. **33**(2), 13-es (2007)
3. Goldsztejn, A.: Modal intervals revisited, part 1: a generalized interval natural extension. Reliable Comput. **16**, 130–183 (2012)
4. Goldsztejn, A.: Modal intervals revisited, part 2: a generalized interval mean value extension. Reliable Comput. **16**, 184–209 (2012)
5. Graillat, S., Jeangoudoux, C., Lauter, C.: MPDI: a decimal multiple-precision interval arithmetic library. Reliable Comput. **25**, 38–52 (2017)
6. Hansen, E.R.: Global Optimization Using Interval Analysis. Marcel Dekker, New York (1992)
7. Heimlich, O.: Interval arithmetic in GNU Octave. In: SWIM 2016: Summer Workshop on Interval Methods, France (2016)
8. IEEE: Institute of Electrical and Electronic Engineers: 754–1985 - IEEE Standard for Binary Floating-Point Arithmetic. IEEE Computer Society (1985)
9. IEEE: Institute of Electrical and Electronic Engineers: 754–2008 - IEEE Standard for Floating-Point Arithmetic. IEEE Computer Society (2008)
10. IEEE: Institute of Electrical and Electronic Engineers: 1788–2015 - IEEE Standard for Interval Arithmetic. IEEE Computer Society (2015)
11. Kahan, W.: How Futile are Mindless Assessments of Roundoff in Floating-Point Computation? (2006). https://people.eecs.berkeley.edu/~wkahan/Mindless.pdf
12. Kaucher, E.: Interval analysis in the extended interval space IR. Comput. Supplementa **2**(1), 33–49 (1980). https://doi.org/10.1007/978-3-7091-8577-3_3
13. Kearfott, R.B., Nakao, M.T., Neumaier, A., Rump, S.M., Shary, S.P., van Hentenryck, P.: Standardized notation in interval analysis. Comput. Technol. **15**(1), 7–13 (2010)
14. Kearfott, R.B.: An overview of the upcoming IEEE P-1788 working group document: standard for interval arithmetic. In: IFSA/NAFIPS, pp. 460–465. IEEE, Canada (2013)
15. Moore, R.E.: Interval Analysis. Prentice Hall, Englewood Cliffs (1966)
16. Moore, R.E.: Methods and Applications of Interval Analysis. SIAM Studies in Applied Mathematics, Philadelphia (1979)
17. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM, Philadelphia (2009)
18. Nadezhin, D.Y., Zhilin, S.I.: Jinterval library: principles, development, and perspectives. Reliable Comput. **19**(3), 229–247 (2014)
19. Nehmeier, M.: libieeep1788: A C++ Implementation of the IEEE interval standard P1788. In: Norbert Wiener in the 21st Century, pp. 1–6, IEEE, Australia (2014)
20. Neumaier, A.: Interval Methods for Systems of Equations. Cambridge University Press, Cambridge (1990)
21. Pryce, J.: The forthcoming IEEE standard 1788 for interval arithmetic. In: Nehmeier, M., Wolff von Gudenberg, J., Tucker, W. (eds.) SCAN 2015. LNCS, vol. 9553, pp. 23–39. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31769-4_3
22. Ratz, D.: Inclusion Isotone Extended Interval Arithmetic. Report 5 (96 pages). Institut für Angewandte Mathematik, Universität Karlsruhe (1996)

23. Revol, N., Denneulin, Y., Méhaut, J.-F., Planquelle, B.: Parallelization of continuous verified global optimization. In: 19th IFIP TC7 Conference on System Modelling and Optimization, Cambridge, United Kingdom (1999)
24. Revol, N., Rouillier, F.: Motivations for an arbitrary precision interval arithmetic and the MPFI library. Reliable Comput. **11**(4), 275–290 (2005)
25. Revol, N.: Introduction to the IEEE 1788-2015 standard for interval arithmetic. In: Abate, A., Boldo, S. (eds.) NSV 2017. LNCS, vol. 10381, pp. 14–21. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63501-9_2
26. Rump, S.M.: Verification methods: rigorous results using floating-point arithmetic. Acta Numerica **19**, 287–449 (2010)
27. Sunaga, T.: Geometry of Numerals. Master thesis, U. Tokyo, Japan (1956)
28. Théveny, P.: Numerical Quality and High Performance In Interval Linear Algebra on Multi-Core Processors. PhD thesis, ENS Lyon, France (2014)
29. Tucker, W.: Validated Numerics - A Short Introduction to Rigorous Computations. Princeton University Press, Princeton (2011)