Numerical Reproducibility and Parallel Computations: Issues for Interval Algorithms

Nathalie Revol, Philippe Théveny

Numerical Reproducibility and Parallel Computations: Issues for Interval Algorithms

Nathalie Revol [1] and Philippe Théveny [2]

Université de Lyon
LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL), ENS de Lyon, France
[1]INRIA - Nathalie.Revol@ens-lyon.fr
WWW home page: http://perso.ens-lyon.fr/nathalie.revol/
[2]ENS de Lyon - Philippe.Theveny@ens-lyon.fr
WWW home page: http://perso.ens-lyon.fr/philippe.theveny/

Abstract

What is called *numerical reproducibility* is the problem of getting the same result when the scientific computation is run several times, either on the same machine or on different machines, with different numbers of processing units, types, execution environments, computational loads etc. This problem is especially stringent for HPC numerical simulations. In what follows, the focus is on parallel implementations of interval arithmetic using floating-point arithmetic. For interval computations, numerical reproducibility is of course an issue for testing and debugging purposes. However, as long as the computed result encloses the exact and unknown result, the inclusion property, which is the main property of interval arithmetic, is satisfied and getting bit for bit identical results may not be crucial. Still, implementation issues may invalidate the inclusion property. Several ways to circumvent these issues are presented, on the example of the product of matrices with interval coefficients.

Keywords: interval arithmetic, numerical reproducibility, parallel implementation, floating-point arithmetic, rounding mode

1 Introduction

Interval arithmetic is a mean to perform numerical computations and to get a guarantee on the computed result. In interval arithmetic, one computes with intervals, not numbers. These intervals are guaranteed to contain the (set of) exact values, both input values and computed values, and this property is preserved throughout the computations. Indeed, the fundamental theorem of interval arithmetic is the *inclusion property*: each computed interval contains the exact (set of) result(s). To learn more about interval arithmetic, see [28, 29, 31, 47].

The inclusion property is satisfied by the definition of the arithmetic operations, and other operations and functions acting on intervals. Implementations of interval arithmetic on computers often rely on the floating-point unit provided by the processor. The inclusion property is preserved via the use of directed roundings. If an interval is given exactly for instance by its endpoints, the left endpoint of its floating-point representation is the rounding

towards $-\infty$ of the left endpoint of the exact interval. Likewise, the right endpoint of its floating-point representation is the rounding towards $+\infty$ of the right endpoint of the exact interval. Similarly, directed roundings are used to implement mathematical operations and functions: in this way, roundoff errors are accounted for and the inclusion property is satisfied.

In this paper, the focus is on the problems encountered while implementing interval arithmetic, using floating-point arithmetic, on multicore architectures. In a sense these issues relate to issues known as problems of *numerical reproducibility* in scientific computing using floating-point arithmetic on emerging architectures. The common points and the differences and concerns which are specific to interval arithmetic will be detailed.

Contributions. The main contributions of this paper are **the identification of problems that cause numerical irreproducibility, and recommendations to circumvent these problems**. The classification proposed here distinguishes between three categories. The first category, addressed in Section 4, concerns problems of variable computing precision that occur both in sequential and parallel implementations, both for floating-point and interval computations. These behaviours are motivated by the quest of speed, at the expense of accuracy on the results, be they floating-point or intervals results. However, even if these behaviours hinder numerical reproducibility, usually they do not threaten the validity of interval computations: interval results satisfy the inclusion property.

The second category, developed in Section 5, concerns the order of operations. Due mainly to the indeterminism in multithreaded computations, the order of operations may vary. It is the most acknowledged problem. Again, this problem is due to the quest of short execution time, again it hinders numerical reproducibility of floating-point or interval computations, but at first sight it does not invalidate the inclusion property. However, an interval algorithm will be presented, whose validity relies on an assumption on the order of operations.

The problems of the third category, detailed in Section 6, are specific to interval computations and have an impact on the validity of interval results. The question is whether directed rounding modes, set by the user or the interval library, are respected by the environment. These problems occur both for sequential and parallel computations. Ignoring the rounding modes can permit to reduce the execution time. Indeed, changing the rounding mode can incur severe penalty: the slowdown lies between 10 and 100 on architectures where the rounding modes are accessed via global flags and where changing a rounding mode implies to flush pipelines. However, it seems that the most frequent motivation is either the ignorance of issues related to rounding modes, or the quest for an easier and faster development of the execution environment by overseeing these issues.

For each category, we give recommendations for the design of interval algorithms: numerical reproducibility is still not attained, but the impact of the problems identified is attenuated and the inclusion property is preserved.

Before detailing our classification, numerical reproducibility is defined in Section 2 and then an extensive bibliography is detailed in Section 3.

2 Problem of Numerical Reproducibility

In computing in general and in scientific computing in particular, the quest for speed has led to parallel or distributed computations, from multithreaded programming to high-performance

computing (HPC). Within such computations, contrary to sequential ones, the order in which events occur is not deterministic. Events here is a generic term which covers communications, threads or processes, modification of shared data structures.... The order in which communications from different senders arrive at a common receiver, the relative order in which threads or processes are executed, the order in which shared data structures such as working lists are consulted or modified, is not deterministic. Indeed, for the example of threads, the quest for speed leads a greedy scheduler to start a task as soon as it is ready, and not at the time or order at which it should start in a sequential execution. One consequence of this indeterminism is that speed is traded against numerical reproducibility of floating-point computations: computed results may differ, depending on the order in which events happened in one run or another. However, numerical reproducibility is a desirable feature, at least for debugging purposes: how is it possible to debug a bug that occurs sporadically, in an indeterministic way? Testing is also impossible without numerical reproducibility: when a program returns results which differ from the results of a reference implementation, nothing can be concluded in the absence of numerical reproducibility. In [42], not only the use of higher computing precision is recommended, but also the use of interval arithmetic or of some variant of it in order to get numerical guarantees, and the use of tools that can diagnose sensitive portions of code and take corrective actions. We follow these tracks and we focus on interval arithmetic, the same conclusions can be drawn for its variants.

First, let us introduce the problem of numerical reproducibility in floating-point arithmetic. Floating-point arithmetic is standardised and the IEEE-754 standard [17, 18] specifies completely the formats of floating-point numbers, their binary representation and the behaviour of arithmetic operations. One of the goals of this standardisation was to enable portability and reproducibility of numerical computations across architectures. However, several runs of the same code on different platforms, or even on the same platform when the code is parallel (typically, multithreaded) may yield different results. Getting the same result on the same platform is sometimes called *repeatability*. Only the word *reproducibility* will be used throughout this paper.

A first explanation is that, on some architectures, and with some programming languages and compilers, intermediate computations such as the intermediate sum in $s \leftarrow a+b+c$ can take place with variable precision. This precision can depend on the availability of extended size registers (such as 80-bits registers on 64-bits architectures), on whether operations use registers or store intermediate results in memory, on data alignment or on the execution of other threads [3], on whether an intermediate computation is promoted to higher precision or not. The impact of the computing precision on numerical irreproducibility is discussed in details in Section 4.

Let us stick to our toy example $s \leftarrow a+b+c$ to illustrate a second cause to numerical irreproducibility. As floating-point addition is not associative (neither is multiplication), the order according to which the operations are performed matters. Let us illustrate this with an example in double precision: $\mathrm{RN}(1 + (2^{100} - 2^{100}))$ (where RN stands for *rounding-to-nearest*) yields 1, since $\mathrm{RN}(2^{100} - 2^{100}) = 0$, whereas $\mathrm{RN}((1 + 2^{100}) - 2^{100})$ yields 0, since $\mathrm{RN}(1+2^{100}) = 2^{100}$. The last result is called a *catastrophic cancellation*: most or all accuracy is lost when close (and even equal in this example) large numbers are subtracted and only roundoff errors remain, hiding the meaningful result. Real-life examples of this problem are to be found in [14], where the results of computations for ocean-atmosphere simulation strongly depends on the order of operations. Other examples in punching of metal sheets are given in [7]. This lack of associativity of floating-point operations implies that the result of a

reduction operation depends on the order according to which the operations are performed. However, with multithreaded or parallel or distributed implementations, this order is not deterministic and the computed result thus varies from one execution to the next. How the order of operations influences numerical computations and interval computations is detailed in Section 5.

A last issue, which is specific to the implementation of interval arithmetic on parallel environments, is the respect of rounding modes. As already mentioned, the implementation of interval arithmetic crucially requires directed rounding modes. However, it has been observed that rounding modes are not respected by numerical libraries such as BLAS [22], nor by compilers when default options are used, nor by execution environment for multithreaded computations, nor by parallel languages such as OpenMP or OpenCL. Either this is not documented, or this is explicitly mentioned as being not supported, cf. Section 6 for a more detailed discussion. Respecting the rounding modes is required by the IEEE-754 standard for floating-point arithmetic and the behaviours just mentioned are either misbehaviours as in the example of Section 6.1, or are (often undocumented) features of the libraries, often "justified" by the quest of shorter execution times.

These phenomena explain the lack of numerical reproducibility for floating-point computations, i.e. the fact that two different runs yields two different results, or the loss of the inclusion property in interval arithmetic, due to the non-respect of rounding modes.

Facing this lack of reproducibility, various reactions are possible. One consists in acknowledging the computation of differing results as an indication of a lack of numerical stability. In a sense, a positive way of considering numerical irreproducibility is to consider it as useful information on the numerical quality of the code. However, for debugging and testing purposes, reproducibility is more than helpful. For such purposes, *numerical reproducibility means getting bitwise identical results from one run to the next.* Indeed, this is the most common definition of numerical reproducibility. This definition is also useful for contractual or legal purposes (architectural design, drug design are instances mentioned in the slides corresponding to [5]), as long as a reference implementation, or at least a reference result, is given. However, this definition is not totally satisfactory as the result is not well-defined. In particular, it says nothing about the accuracy of the result. Requiring the computed result to be the correct rounding of the exact result is a semantically meaningful definition of numerical reproducibility. The computed result is thus uniquely defined. The difficulty with this definition is to devise an algorithm that is efficient on parallel platforms and that computes the correct rounding of the exact result. This definition has been adopted in [25], for LHC computations of 6000,000 jobs on 60,000 machines: efficient mathematical functions such as exponential and logarithm, with correct rounding, were available. However, in most cases it is not known how to compute efficiently the correctly rounded result. For this reason, this definition of numerical reproducibility as computing the correct rounding of the exact result may be too demanding. Thus we prefer to keep separate the notions of numerical reproducibility (i.e. getting bitwise identical results) and of correct rounding.

To sum up, numerical reproducibility can be defined as getting bitwise identical results, where these results have to been specified in some way, e.g. by a reference implementation which can be slow. We consider it is simpler not to mix this notion with the notion of correct rounding of the exact result, which is uniquely specified. Our opinion has been reinforced by

the results in [5]: it is even possible to define several levels of numerical reproducibility, each one corresponding to a level of accuracy. One has thus a hierarchy of reproducibilities, corresponding to different tradeoffs between efficiency and accuracy, ranging from low numerical quality to correct rounding.

3 Previous Work

In the previous section, we introduced various sources of numerical irreproducibility and we delineated their main features. More detailed and technical explanations are given in [12]: implementation issues such as data race, out-of-order execution, message buffering with insufficient buffer size, non-blocking communication operations are also introduced. A tentative classification of indeterminism can be found in [3], it distinguishes between *external determinism* that roughly corresponds to getting the same results independently of the internal states reached during the computation, and *internal determinism* that requires that internal execution steps are the same from one run to the other. External indeterminism can be due to data alignment that varies from one execution to the next, order of communications with "wildcard receives" of MPI, and other causes already mentioned. In what follows, only external indeterminism will be considered.

Numerical irreproducibility, in particular of summations of floating-point numbers, is cited as early as 1994 in [20] for weather forecasting applications. More recently, it has been clearly put in evidence in [7] where the application is the numerical simulation of a deep drawing process for deforming metal sheets: depending on the execution, the computed variations of the thickness of the metal sheet vary. Other references mentioning numerical irreproducibility are for instance [14] for an application in ocean-atmosphere simulation, [23] and reference [10] herein about digital breast tomosynthesis, or [48] for power state estimation in the electricity grid.

3.1 The Example of the Summation

The non-associativity of floating-point is the major explanation to the phenomenon of numerical irreproducibility. The simplest problem that exemplifies the non-associativity is the summation of n floating-point numbers. It is also called a reduction of n numbers with the addition operation. Not surprisingly, many efforts to counteract numerical irreproducibility focus on the summation problem, as accuracy can be lost due to catastrophic cancellation or catastrophic absorption[1] in summation. Let us list some of them in chronological order of publication. An early work [14] on the summation uses and compares several techniques to get more accuracy on the result: the conclusion is that compensated summation and the use of double-double arithmetic give the best results. Following this work, for physical simulations, conservation laws were numerically ensured using compensated sums, either Kahan's version or Knuth's version, and implemented as MPI reduction operators in [39]. Similarly, in [43], a simplified form of "single-single" arithmetic is employed on GPU to sum the energy

[1] Catastrophic absorption occurs in a sum when small numbers are added to a large number and "disappear" in the roundoff error, i.e. they do not appear in the final result, whereas adding first these small numbers would result in a number large enough to be added to the first large number and be "visible".

and satisfy numerically the conservation law. Bailey, the main author of QD, a library for double-double and quad-double arithmetic [15], also advocates the use of this higher-precision arithmetic in [1]. However, even if the use of extra computing precision yields accurate results for more ill-conditioned summations, which means the obtention of results with more correct bits, it does not ensure numerical reproducibility, as solving even worse-conditioned problems shows. Since the problem can be attributed to the variations in the summation order, a first approach to get reproducible results consists in fixing the reduction tree [48]. More precisely, using a fixed integer K, the array of summands is split into K chunks of consecutive subarrays, each subarray (or chunk) being summed sequentially (but each chunk can be summed independently of the other ones) and then the order of the reduction of the K partial sums is also sequential.

Another approach in [5, 6] consists in what is called *pre-rounding*. Even if the actual implementation is very different from the process explained here, it can be thought of as a sum in fixed-point, as learnt in elementary school. The mantissa of every summand is aligned with respect to the point, then a leftmost "vertical slice" is considered and added to produce a sum S_1. The width of this slice is chosen in such a way that the sum S_1 can be performed exactly using the width of the mantissa of floating-point numbers (e.g. 53 bits for the double precision format). As this sum S_1 is exact, it is independent of the order in which the additions are performed and is thus numerically reproducible on any platform. To get a more accurate result, a second vertical slice, just right to the first one, can be summed, again yielding an exact sum S_2 and the final result is the (floating-point, thus inexact) sum of S_1 and S_2. An increase of the number K of slices corresponds to an increase of the computing precision. However, for a fixed K, each partial sum $S_1, \ldots, S_K$ is exact and thus numerically reproducible. The details to determine the slices and to get $S_1, \ldots S_K$ are given in [5] and a faster algorithm for exascale computing, i.e. for really large platforms, is provided in [6].

3.2 Approaches to Reach Numerical Reproducibility

In [21] a tool, called MARMOT, that detects and signals race conditions and deadlocks in MPI codes is introduced, but reduction operations are not handled.

The original proposal for Java by Gosling [10] included numerical reproducibility of floating-point computations. To reach reproducibility, it prohibited the use of any format different from Binary32 and Binary64, the use of the FMA as well as any optimisation based on the associativity of the operators. It also forbade changes of the rounding modes [33] and the only rounding mode is to-nearest [11, Section 4.2.4]. It did not include the handling of exceptions via flags, as required by the IEEE-754 standard. The seminal talk by Kahan in 1998 [19] has shaken these principles. Actually Kahan disputed mainly the lack of exception handling. It seems that this dispute opened the door to variations around Java. Indeed Java Grande [34, 44] proposes to allow the use of longer formats and of FMA. The use of the associativity of operations to optimise the execution time has been under close scrutiny for a longer lapse of time and remains prohibited [11, Section 15.7.3], as explicit rules are given, e.g. left-to-right priority for $+$, $-$, $\times$. However, strict adherence to the initial principles of Java can be enforced by using the `StrictFp` keyword. For instance, the VSEit environment for modelling complex systems, simulating them and getting a graphical view [2], uses the `StrictMath` option in Java as a way to get numerical reproducibility. However, the need for getting reproducibility is not explained in much details.

Finally, Intel MKL (Math Kernel Library) 11.0 introduces a feature called Conditional

Numerical Reproducibility (CNR) [46] which provides functions for obtaining reproducible floating-point results. When using these new features, Intel MKL functions are designed to return the same floating-point results from run-to-run, subject to the following limitations:

- calls to Intel MKL occur in a single executable
- input and output arrays in function calls must be aligned on 16, 32, or 64 byte boundaries on systems with SSE / AVX1 / AVX2 instructions support (resp.)
- the number of computational threads used by the library remains constant throughout the run.

These conditions are rather stringent. Another approach to numerical reproducibility consists in providing correctly rounded functions, at least for the mathematical library [25].

The approaches presented here are not yet entirely satisfactory, either because they do not really offer numerical reproducibility or because they handle only summation, or because performances are too drastically slowed down. Furthermore, none addresses interval computations. In what follows, we propose a classification of the sources of numerical irreproducibility for interval computations and some recommendations to circumvent these problems, even if we do not have the definitive solution. In our classification, a first source of problem is the variability of the employed computing precision.

4 Computing Precision

4.1 Problems for Floating-Point Computations

The computing precision used for floating-point computations depends on the employed format (as defined by IEEE-754 standard [18]). The single precision corresponds to the representation of floating-point numbers on 32 bits, where 1 bit is used for the sign, 8 bits for the exponent and the rest for the significand. The corresponding format is called `Binary32`. The double precision uses 64 bits, hence the name `Binary64`, with 1 bit for the sign and 11 for the exponent. The quadruple precision, also known as `Binary128`, uses 128 bits, with 1 bit for the sign and 15 bits for the exponent. The rest of this Section owes much to [8].

On some architectures (IA32 / x87), registers have a longer format: 80 bits instead of 64. The idea prevailing to the introduction of these long registers was to provide higher accuracy by computing, for a while, with higher precision than the Binary64 format and to round the result into a Binary64 number only after several operations. However it entailed the so-called "double-rounding" problem: an intermediate result is first rounded into a 80-bits number, then into a 64-bits number when it is stored, but these two successive roundings can yield a result different from rounding directly into a 64-bits number. From the point of view of numerical reproducibility, the use of extended precision registers is also troublesome, as the operations which take place within registers and the temporary storage into memory can occur at different stages of the computation, they may vary from run to run, depending for instance on the load of the current processor, on data alignment or on the execution of other threads [3].

Another issue is the format chosen for the intermediate result, say for the intermediate sums in the expression $a+b+c+d$, where a, b, c and d are Binary32 floating-point formats. On SSE2 or AVX architectures, $(a+b)+(c+d)$ will execute faster than $((a+b)+c)+d$, as the two additions $a+b$ and $c+d$ can be performed in parallel. Notwithstanding the order in

which the intermediate sums are computed, let us focus on the problem of the precision. If the architecture offers Binary64 and if the language is C or Python, then these intermediate sums may be promoted to Binary64. It will not be the case if the language is Java with the `StrictFp` keyword, or Fortran. In C, it may or may not be the case, depending on the compiler and on the compilation options: the compiler may prefer to take advantage of a vectorised architecture like SSE2 or AVX, where 2 Binary32 floating-point additions are performed in parallel, in the same amount of time as 1 Binary64 floating-point addition. The compiler may also decide to use more accurate registers (64 bits or 80 bits) when such vectorised devices are not available. Thus, depending on the programming language, on the compiler and its options and on the architecture, the intermediate results may vary, as does the final result. In some languages, it is possible to gain a posteriori some knowledge on the employed precision. In C99, the value of `FLT_EVAL_METHOD` gives an indication, at run-time, on the intermediate chosen format: indeterminate, double or long double.

4.2 Problems for Interval Computations

The notion of precision in interval arithmetic could be regarded as the radius of the input arguments. It is known that the overestimation of the result of a calculation is proportional to the radius of the input interval, with a proportionality constant which depends on the computed expression. More precisely [31, Section 2.1], if f is a Lipschitz-continuous function: $D \subset \mathbb{R}^n \to \mathbb{R}$, then

$$q(\mathbf{f}(\mathbf{x}), f(\mathbf{x})) = \mathcal{O}(r) \text{ if } \mathrm{rad}(\mathbf{x}) \leq r$$

where boldface letters denote interval quantities, $f(\mathbf{x})$ is the exact range of f over the interval $\mathbf{x}$, $\mathbf{f}(\mathbf{x})$ is an overestimation of $f(\mathbf{x})$ computed from an expression for f using interval arithmetic in a straightforward way, and q stands for the Hausdorff distance. As the first interval here encloses the second, $q(\mathbf{f}(\mathbf{x}), f(\mathbf{x}))$ is simply $\max(\inf(f(\mathbf{x})) - \inf(\mathbf{f}(\mathbf{x})), \sup(\mathbf{f}(\mathbf{x})) - \sup(f(\mathbf{x})))$. In this formula, r can be considered as the precision. It is possible to improve this result by using more elaborate approaches to interval arithmetic, such as a Taylor expansion of order 1 of the function f. Indeed, if f is a continuously differentiable function then [31, Section 2.3],

$$q(\mathbf{f}(\mathbf{x}), f(\mathbf{x})) = \mathcal{O}(r^2) \text{ if } \mathrm{rad}(\mathbf{x}) \leq r$$

where $\mathbf{f}(\mathbf{x})$ is computed using a so-called centered form. As f is Lipschitz, it holds that the radius of $f(\mathbf{x})$ is also proportional to the radius of the input interval. In other words, the accuracy on the result improves with the radius, or precision, of the input. These results hold for an underlying exact arithmetic.

This result in interval analysis can be seen as an equivalent to the rule of thumb in numerical floating-point analysis [16, Chapter 1]. Considered with optimism, this means that it suffices to increase the computing precision – in floating-point arithmetic – or the precision on the inputs – in interval arithmetic – to get more accurate results. In interval arithmetic, this can be done through bisection of the inputs, to get tight intervals as inputs and to reduce r in the formula above. The final result is then the union of the results computed for each subinterval. A more pragmatic point of view is first that, even if the results are getting more and more accurate as the precision increases, there is still no guarantee that the employed precision will allow to reach a prescribed accuracy for the result. Second, reproducibility is

still out of reach, as developed in Section 5.

In this paper, we consider interval arithmetic implemented using floating-point arithmetic. Thus the discussion above does not account for the limited precision of the underlying floating-point arithmetic. Indeed, computed interval results also suffer from the problems of floating-point arithmetic, namely the possible loss of accuracy. Even if directed roundings make it possible to ensure the inclusion property, there is no information and no guarantee about the tightness of the computed interval.

4.3 Recommendations

We advocate the use of the mid-rad representation on the one hand, and the use of iterative refinement on the other hand. The mid-rad representation $< m, r >$ corresponds to the interval $[m - r, m + r] = \{x \ : \ |m - x| \leq r\}$. An advantage of the mid-rad representation is that thin intervals are represented more accurately in floating-point arithmetic. For instance, let us consider $r = 1/2\text{ulp}(m)$, i.e. r is a power of 2 that corresponds, roughly speaking, to half the last bit in the floating-point representation of m. Then the floating-point representation by endpoints of the interval is $[\text{RD}(m - r), \text{RU}(m + r)]$ which is $[m - 2r, m + 2r]$ i.e. in this example the width of the interval is doubled with the representation by endpoints. The reader has to be aware that no representation, neither mid-rad nor by endpoints, always supersedes the other one.

We also recommend the use of iterative refinement where applicable. Indeed, even if the computations of the midpoint and the radius of the result suffer from the aforementioned lack of accuracy, iterative refinement (usually a few iterations suffice) recovers a more accurate result from this inaccurate one.

Let us illustrate this procedure on the example of linear system solving $\mathbf{Ax} = \mathbf{b}$ (cf. [31, Chapter 4], [29, Chapter 7] for an introduction). Once an initial approximation $\mathbf{x^0}$ is computed, the residual $\mathbf{r} = \mathbf{b} - \mathbf{Ax^0}$ is computed *using twice the current precision* (and here we rejoin the solutions in [1, 14, 39, 43] already mentioned), as much cancellation occurs in this calculation. Then, solve – again approximately – the linear system with the same matrix $\mathbf{Ae} = \mathbf{r}$ (and use every pre-computations done on $\mathbf{A}$) and correct $\mathbf{x^0}$: $\mathbf{x^1} \leftarrow \text{mid}(\mathbf{x^0}) + \mathbf{e}$. Under specific assumptions, but independently of the order of the operations, it is possible to relegate the effects of the intermediate precision and of the condition number after the significant bits in the destination format [32]. In other words, the overestimation due to the floating-point arithmetic is minimal: only one ulp (or very few ulps) on the precision of the interval result. (Let us recall [30] that for $x \in \mathbb{R}$, $x \in [2^e, 2^{e+1})$ then $\text{ulp}(x) = 2^{e-p+1}$ in radix-2 floating-point arithmetic with p bits used for the significand and that $\text{ulp}(-x) = \text{ulp}(x)$ for negative x.)

Another study [45] also takes into account the effect of floating-point arithmetic. It suggests that the same approach applies to nonlinear system solving and that the iterative refinement, called in this case Newton iteration, again yields fully accurate results, i.e. up to 1 ulp of the exact result.

However, in both cases it is assumed that enough steps have been performed: if there is a limit on the number of steps, then one run could converge but not the other one and again numerical reproducibility would not be gained.

To conclude on the impact of the computing precision: it raises no problem for the validity of interval computations, i.e. the inclusion property is satisfied, but it influences the accuracy

of the result and the execution time. It seems that the same could be said for the order of the operations, which is the issue discussed next, but it will be seen that the validity of interval computations can depend on it.

5 Order of the Operations

5.1 Problems for Interval Computations

As already abundantly mentioned, a main explanation to the lack of reproducibility of floating-point computations is the lack of associativity of floating-point operations (addition, multiplication).

Interval arithmetic also suffers from a lack of algebraic properties. In interval arithmetic, the square operation differs from the multiplication by the same argument, because variables (x and y in the example) are decorrelated:

$$\begin{array}{rcccl} & [-1,2]^2 & = & \{x^2,\ x \in [-1,2]\} & = & [0,4] \\ \neq & [-1,2]\cdot[-1,2] & = & \{x\cdot y,\ x \in [-1,2],\ y \in [-1,2]\} & = & [-2,4]. \end{array}$$

This problem is often called *variable dependency*.

In interval arithmetic, the multiplication is not distributive over the addition, again because of the decorrelation of the variables.

Finally, interval arithmetic implemented using floating-point arithmetic suffers from all these algebraic features. In any case, the computed result contains the exact result. However, it is not possible to guarantee that one order produces tighter results than another one, as illustrated by the following example. In the Binary64 format, the smallest floating-point number larger than 1 is $1+2^{-52}$. Let us consider three intervals with floating-point endpoints: $\mathbf{A}_1 = [-2^{-53}, 2^{-52}]$, $\mathbf{A}_2 = [-1, 2^{-52}]$ and $\mathbf{A}_3 = [1, 2]$. Using the representation by endpoints and the implementation in floating-point arithmetic of $[\underline{a}, \bar{a}] + [\underline{b}, \bar{b}]$ as $[\mathrm{RD}(\underline{a}+\underline{b}), \mathrm{RU}(\bar{a}+\bar{b})]$ where RD denotes the rounding mode towards $-\infty$ or rounding downwards and RU denotes the rounding mode towards $+\infty$ or rounding upwards, one gets for $(\mathbf{A}_1 + \mathbf{A}_2) + \mathbf{A}_3$:

$$\begin{array}{lcccl} \mathrm{tmp}_1 & := & \mathbf{A}_1 + \mathbf{A}_2 & = & [\mathrm{RD}(-2^{-53}-1), \mathrm{RU}(2^{-52}+2^{-52})] \\ & & & = & [-1-2^{-52}, 2^{-51}] \\ \text{and finally} & & & & \\ \mathbf{B}_1 & := & \mathrm{tmp}_1 + \mathbf{A}_3 & = & [\mathrm{RD}(-1-2^{-52}+1), \mathrm{RU}(2^{-51}+2)] \\ & & & = & [-2^{-52}, 2+2^{-51}]. \end{array}$$

And one gets for $\mathbf{A}_1 + (\mathbf{A}_2 + \mathbf{A}_3)$:

$$\begin{array}{lcccl} \mathrm{tmp}_2 & := & \mathbf{A}_2 + \mathbf{A}_3 & = & [\mathrm{RD}(-1+1), \mathrm{RU}(2^{-52}+2)] \\ & & & = & [0, 2+2^{-51}] \\ \text{and finally} & & & & \\ \mathbf{B}_2 & := & \mathbf{A}_1 + \mathrm{tmp}_2 & = & [\mathrm{RD}(-2^{-53}+0), \mathrm{RU}(2^{-52}+2+2^{-51})] \\ & & & = & [-2^{-53}, 2+2^{-50}]. \end{array}$$

The exact result is $\mathbf{B} := [2^{-53}, 2+2^{-51}]$ and this interval is representable using floating-point endpoints. It can be observed that both $\mathbf{B}_1$ and $\mathbf{B}_2$ enclose $\mathbf{B}$: the inclusion property is satisfied. Another observation is that $\mathbf{B}_1$ overestimates $\mathbf{B}$ to the left and $\mathbf{B}_2$ to the right. Of

course, one can construct examples where both endpoints are under- or over-estimated.

Not only does the order in which non-associative operations such as additions are performed matter, but other orders do as well. Let us go back to the bisection process mentioned in Section 4.2. Indeterminism is present in the bisection process, and it introduces more sources of irreproducibility. Indeed, with bisection techniques, one interval is split into 2 and one half (or the two halves) are stored in a list for later processing. The order in which intervals are created and processed is usually not deterministic, as the list is used by several threads. This can even influence the creation or bisection of intervals later in the computation. Typically, in algorithms for global optimisation [13, 35], the best enclosure found so far for the optimum can lead to the exploration or destruction of intervals in the list. As the order in which intervals are explored varies, this enclosure for the optimum varies and thus the content of the list varies – and not only the order in which its content is processed.

5.2 Recommendations

As shown in the example of the sum of three intervals, even if the result depends on the order of the operations, the inclusion property holds and it always encloses the exact results. Thus an optimistic view is that numerical reproducibility is irrelevant for interval computations, as the result always satisfies the inclusion property. An even more optimistic view could be that getting different results is good, since the sought result lies in their intersection. Intersecting the various results would yield even more accuracy. Indeed, with the example above, the exact result $\mathbf{B}$ is recovered by intersecting $\mathbf{B}_1$ and $\mathbf{B}_2$. Unfortunately this is rarely the case, and usually no computation yields tighter results than the other ones. Thus the order may matter.

We have already mentioned the benefit of using the mid-rad representation in Section 4.3. Another, much more important, advantage is to be able to benefit from efforts made in developing mathematical libraries. As shown by Rump in his pioneering work [40], in linear algebra in particular, it is possible to devise algorithms that use floating-point routines. Typically these algorithms compute the midpoint of the result using optimised numerical routines, and they compute afterwards the radius of the result using again optimised numerical routines. Let us illustrate this approach with the product of matrices with interval coefficients, given in [40]: $\mathbf{A} =< A_m, A_r >$ and $\mathbf{B} =< B_m, B_r >$ are matrices with interval coefficients, represented as matrices of midpoints and matrices of radii. The product $\mathbf{A} \cdot \mathbf{B}$ is enclosed in $\mathbf{C} =< C_m, C_r >$ where an interval enclosing C_m is computed as $[\mathrm{RD}(A_m \cdot B_m), \mathrm{RU}(A_m \cdot B_m)]$ using optimised BLAS3 routines for the product of floating-point matrices. Then C_r is computed using $\mathrm{RU}((|A_m| + A_r) \cdot B_r + A_r \cdot |B_m|)$, again using optimised BLAS routines. In [40], the main benefit which is announced is the gain in execution time, as these routines are really well optimised for a variety of architectures, e.g. in GOTO-BLAS or Atlas or in MKL, the library developed by Intel for its architectures. We also foresee the benefit of using reproducible libraries, once they are developed, such as the CNR version of Intel MKL, and once their usage and performance are optimised.

To sum up, so far only accuracy and speed of interval computations can be affected by the order of operations, but not the validity of the results. As discussed now, the order may also impact the validity of interval results, i.e. the inclusion property may not be satisfied.

5.3 Concerns Regarding the Validity of Interval Results

Apart from their lack of numerical reproducibility, there is another limit to the current usability of floating-point BLAS routines for interval computations. This limit lies in the use of strong assumptions on the order in which operations are performed. Let us again exemplify this issue on the product of matrices with interval coefficients. The algorithm given above reduces interval operations to floating-point matrices operations. However, it requires 4 calls to BLAS3 routines. An algorithm has recently been proposed [41] that requires only 3 calls to floating-point matrix products. An important assumption to ensure the inclusion property is that these 2 floating-point matrix products perform the basic arithmetic operations in the same order. Namely, the algorithm relies on the following theorem [41, Theorem 2.1]: if A and b are two $n \times n$ matrices with floating-point coefficients, if $C = \mathrm{RN}(A \cdot B)$ is computed in any order and if $\Gamma = \mathrm{RN}(|A| \cdot |B|)$ is computed *in the same order*, then the error between C and the exact product $A \cdot B$ satisfies

$$|\mathrm{RN}(A \cdot B) - A \cdot B| \leq \mathrm{RN}((n+2)u\Gamma + \eta)$$

where u is the unit roundoff and η is the smallest normal positive floating-point number ($u = 2^{-52}$ and $\eta = 2^{-1022}$ in Binary64). This assumption on the order is not guaranteed by any BLAS library we know, and the discussion above tends to show that this assumption does not hold for multithreaded computations.

Two solutions can be proposed to preserve the efficiency of this algorithm along with the inclusion property. A straightforward solution consists in using a larger bound for the error on the matrix product, a bound which holds whatever the order for the computations of C and Γ. Following [16, Chapter 2] and barring underflow, a bound of the following form can be used instead [38]:

$$|\mathrm{RN}(A \cdot B) - A \cdot B| \leq \mathrm{RN}\left((1+3u)^n \cdot (\{(1+2u)|A|\} \cdot \{(1+2u)|B|\})\right)$$

A more demanding solution, implemented in [37], consists in implementing simultaneously the two matrix products, in order to ensure the same order of operations. Optimisations to get performances are done: block products to optimise the cache usage (Level 1) and hand-made vectorisation of the code yield performances, even if the performances of well-known BLAS are difficult to attain. Furthermore, this hand-made implementation has a better control on the rounding modes than the BLAS routines, and this is another major point in satisfying the inclusion property, as discussed in the next Section. The lack of respect of the rounding modes is another source of interval invalidity (to paraphrase the term "numerical reproducibility") and it is specific to interval computations.

6 Rounding Modes

6.1 Problems for Interval Computations

Directed rounding modes of floating-point arithmetic are crucial for a correct implementation of interval arithmetic, i.e. for an implementation that preserves the inclusion property. This is a main difference between interval computations and floating-point computations, that usually employ rounding-to-nearest only. It is also a issue not only for reproducibility, but

even for correctness of interval computations, especially for parallel implementations, as it will be shown below.

For some floating-point computations, such as compensated sums mentioned in Section 3, the already mentioned QD library [15], or for XBLAS, a library for BLAS with extended precision [24], the only usable rounding mode is rounding-to-nearest, otherwise these libraries fail to deliver results in higher precision.

For interval arithmetic, both endpoints and mid-rad representations require directed rounding modes. However, many obstacles are encountered. First, it may happen that the compiler is too eager to optimise a code to respect the required rounding mode. For instance, let us compute the enclosure of a/b where both a and b are floating-point numbers, using the following piece of pseudo-code:

```
set_rounding_mode (downwards)
left := a/b
set_rounding_mode (upwards)
right := a/b
```

In order to spare a division which is a relatively expensive operation, a compiler may assign `right := left`, but then the result is not what expected. This example is given in the gcc bug report #34678 entitled *Optimization generates incorrect code with -frounding-math options*. Even using the right set of compilation options does not solve the problem Usually these options, when doing properly the job, counteract optimisations done by the compiler and the resulting code may exhibit poor performances.

Second, interval computations may rely on floating-point routines, such as the BLAS routines in our example of the product of matrices with interval coefficients. For interval computations, it is desirable that the BLAS library respects the rounding mode set before the call to a routine. (However C99 seems to exclude that external libraries called from C99 respect the rounding mode in use, on the contrary C99 allows them to set and modify the rounding mode.) This desirable behaviour is not documented for the libraries we know of, as developed in [22]. The opposite has been observed by A. Neumaier and years later by F. Goualard in MatLab, as explained in his message to `reliable_computing@interval.louisiana.edu` on 29 March 2012 quoted below.

> In a 2008 mail from Pr. Neumaier to Cleve Moler forwarded to the IEEE 1788 standard mailing list (`http://grouper.ieee.org/groups/1788/email/msg00204.html`), it is stated that MATLAB resets the FPU control word after each external call, which would preclude any use of fast directed rounding through some C MEX file calling, say, the fesetround() function.
>
> According to my own tests, the picture is more complicated than that. With MATLAB R2010b under Linux/64 bits I am perfectly able to switch the rounding direction in some MEX file for subsequent computation in MATLAB. The catch is that the rounding direction may be reset by some later events (calling some non-existent function, calling an M-file function, ...). In addition, I cannot check the rounding direction using fegetround() in a C MEX file because it always returns that the rounding direction is set to nearest/even even when I can check by other means that it is not the case in the current MATLAB environment.

A more subtle argument against the naive use of mathematical libraries is based on the monotony of the operations. Let us take again the example of a matrix product, with nonneg-

ative entries, such as in Γ introduced in Section 5.2. If the rounding mode is set to rounding upwards for instance, then it suffices to compute $\Gamma_{i,j}$ as $\mathrm{RU}(\sum_k |a_{i,k}| \cdot |b_{k,j}|)$ to get an overestimation of $\sum_k |a_{i,k}| \cdot |b_{k,j}|$. However, if Γ is computed using Strassen's formulae, then terms of the form $x - z$ and $y + z$ are introduced. To get an upper bound on these terms, one needs an overestimation $\bar{x}$ of x, $\bar{y}$ of y and $\bar{z}$ of z, but also an underestimation $\underline{z}$ of z. The overestimation of $x - z$ can thus be computed as $\mathrm{RU}(\bar{x} - \underline{z})$ and the overestimation of $y + z$ as $\mathrm{RU}(\bar{y} + \bar{z})$. In other words, one may need both over- and under-estimation of intermediate values, i.e. one may need to compute many intermediate quantities twice. This would ruin the fact that Strassen's method is a fast method. Anyway, if a library implementing Strassen's product is called with the rounding mode set to $+\infty$ and respects it, it simply performs all operations with this rounding mode and there is no guarantee that the result overestimates the exact result. To sum up, there is little evidence and little hope that mathematical libraries return an overestimation of the result when the rounding mode is set to $+\infty$.

Third, the programming language may or may not support changes of the rounding mode. OpenCL does not support it, as explicitly stated in the documentation (from the OpenCL Specification Version: 1.2, Document Revision: 15): *Round to nearest even is currently the only rounding mode required by the OpenCL specification for single precision and double precision operations and is therefore the default rounding mode. In addition, only static selection of rounding mode is supported. Dynamically reconfiguring the rounding modes as specified by the IEEE 754 spec is unsupported.* OpenMP does not support it either, as less explicitly stated in the documentation (from OpenMP Application Program Interface, Version 4.0 - RC 2 - March 2013, Public Review Release Candidate 2): *This OpenMP API specification refers to ISO/IEC 1539-1:2004 as Fortran 2003. The following features are not supported: IEEE Arithmetic issues covered in Fortran 2003 Section 14 [...]* which are the issues related to rounding modes.

Fourth, the execution environment, i.e. the support for multithreaded execution, usually does not document either how the rounding modes are handled. For architectures or instructions sets which have the rounding mode specified in the code for the operation, as CUDA for GPU, or as IA64 processors but without access from a high-level programming language, rounding modes are handled smoothly, even if they are accessed with more or less ease. For other environments where the rounding modes are set via a global flag, it is not clear how this flag is handled: it is expected that it is properly saved and restored when a thread is preempted or migrated, it is not documented whether concurrent threads on the same core "share" the rounding mode or whether each of them can use its own rounding mode. To quote [22]: *How a rounding mode change is propagated from one thread or node of a cluster to all others is unspecified in the C standard. In MKL the rounding mode can be specified only in the VML (Vector Math Library) part and any multi-threading or clustering behavior is not documented.*

After this discussion, it may appear utopian to rely too much on directed rounding modes to ensure that the inclusion property is satisfied.

6.2 Recommendations

Our main recommendation is to use bounds on roundoff errors rather than using directed rounding modes, *when it is not safe to do so.* These bounds must be computable using floating-point arithmetic. They must be computed using with rounding-to-nearest which is

most likely to be in use. They are preferably independent of the rounding mode. For instance, the bound given in Section 5.3 can be made independent of the rounding mode by replacing u, which is the roundoff unit for rounding-to-nearest, by $2u$ which is an upper bound for any rounding mode [38]. Another example of this approach to floating-point roundoff errors is the COSY library [36]. This approach is rather brute force, but it is robust to the change of rounding mode.

7 Conclusion

As developed in the preceding sections, obtaining numerical reproducibility is a difficult task. Getting it can severely impede performances in terms of execution time. It is thus worth checking whether numerical reproducibility is really needed, or whether getting guarantees on the accuracy of the results suffices. For instance, as stated in [14] about climate models: *It is known that there are multiple stable regions in phase space* [...] *that the climate system could be attracted to. However, due to the inherent chaotic nature of the numerical algorithms involved, it is feared that slight changes during calculations could bring the system from one regime to another.* In this example, qualitative information, such as the determination of the attractor for the system under consideration, is more important than reproducibility. This questioning goes further in a talk by Dongarra, similar to [9], where he advocated the quest for a guaranteed accuracy and the use of small computing precision, such as single precision, for speed and energy-consumption reasons.

However, numerical reproducibility may be mandatory. In such a case, our main recommendation to conclude this work is the following methodology.

- Firstly, develop interval algorithms that are based on well-established numerical bricks, so as to benefit from their optimised implementation.
- Second, convince developers and vendors of these bricks to clearly specify their behaviour, especially what regards rounding modes.
- If the second step fails, replicate the work done for the optimisation of the considered numerical bricks, to adapt them to the peculiarities and requirements of the interval algorithm. A precursor to this recommendation is the recommendation in [26] that aims at easing such developments: *In order to achieve near-optimal performance, library developers must be given access to routines or kernels that provide computational- and utility-related functionality at a lower level than the customary BLAS interface.* This would make possible to use the lower level bricks used in high-performance BLAS, e.g. computations at the level of the blocks and not of the entire matrix.
- Get free from the rounding mode by bounding, roughly but robustly, errors with formulas independent of the rounding mode if needed.

Eventually, let us emphasise that the problem of numerical reproducibility is different from the more general topic called *Reproducible research in computer science*, which is for instance developed in a complete issue of the *Computing in Science & Engineering* magazine [4]. Reproducible research corresponds to the possibility to reproduce computational results by keeping track of the code version, compiler version and options, input data... used to

produce the results that are often only summed up, mostly as figures, in papers. A possible solution is to adopt a "versioning" system not only for code files, but also for compilation commands, data files, binary files... However, floating-point issues are usually not considered in the related publications; they very probably should.

Acknowledgements. This work is partly funded by the HPAC project of the French Agence Nationale de la Recherche (ANR 11 BS02 013).

References

[1] D. H. Bailey. High-Precision Computation and Reproducibility. In *Reproducibility in Computational and Experimental Mathematics*, Providence, RI, USA, December 2012. `http://www.davidhbailey.com/dhbtalks/dhb-icerm.pdf`

[2] K.-H. Brassel. Advanced Object-Oriented Technologies in Modeling and Simulation: the VSEit Framework. In *ESM 2001 European Simulation Multiconference 2001*, pages 154–160, Prague, Czech Republic, June 2001.

[3] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, and G. L. Lee. Determinism and Reproducibility in Large-Scale HPC Systems (7 pages). In *WODET 2013 - 4th Workshop on Determinism and Correctness in Parallel Programming*, Washington, USA, March 2013. `http://wodet.cs.washington.edu/wp-content/uploads/2013/03/wodet2013-final12.pdf`

[4] S. Foeml and J. F. Claerbout (editors). Reproducible Research. Special issue of *Computing in Science & Engineering*, 11(1), January/February 2009.

[5] J. Demmel and H. D. Nguyen. Fast Reproducible Floating-Point Summation. In *ARITH 2013*, pages 163–172, Austin, TX, USA, April 2013. `http://www.eecs.berkeley.edu/~hdnguyen/public/papers/ARITH21_Fast_Sum.pdf`

[6] J. Demmel and H. D. Nguyen. Numerical Reproducibility and Accuracy at ExaScale. In *ARITH 2013*, Austin, TX, USA, April 2013. `http://www.arithsymposium.org/papers/paper_33.pdf`

[7] K. Diethelm. The Limits of Reproducibility in Numerical Simulation. *Computing in Science & Engineering*, pages 64–71, 2012.

[8] F. de Dinechin. De calculer juste à calculer au plus juste (in English except the title). In *School on Accuracy and Reproducibility of Numerical Computations*, Fréjus, France, March 2013. `http://calcul.math.cnrs.fr/IMG/pdf/2013-Dinechin-PRCN.pdf`

[9] J. Dongarra. Algorithmic and Software Challenges when Moving Towards Exascale. In *ISUM 2013: 4th International Supercomputing Conference in Mexico*, March 2013. `http://www.netlib.org/utk/people/JackDongarra/SLIDES/isum0213.pdf`

[10] J. Gosling. *Sun Microsystems Laboratories - The First Ten Years - 19912001*, chapter Java: an Overview. J. Treichel and M. Holzer, 1995.

[11] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java® Language Specification - Java SE 7 Edition*. Addison Wesley, 2013.

[12] W. D. Gropp. Issues in Accurate and Reliable Use of Parallel Computing in Numerical Programs. In Bo Einarsson, editor, *Accuracy and Reliability in Scientific Computing*, pages 253–263. SIAM, 2005.

[13] E.R. Hansen and G.W. Walster. *Global optimization using interval analysis*. 2nd edition, Marcel Dekker, 2003.

[14] Y. He and C.Q. Ding. Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications. *The Journal of Supercomputing*, 18:259–277, 2001.

[15] Y. Hida, X. S. Li, and D. H. Bailey. Quad Double computation package, 2003-2012. `http://crd-legacy.lbl.gov/~dhbailey/mpdist/`

[16] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. 2nd edition, SIAM, 2002.

[17] Institute of Electrical and Electronic Engineers and American National Standards Institute. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985,* New York, 1985.

[18] Institute of Electrical and Electronic Engineers. IEEE standard for floating-point arithmetic. *IEEE Standard 754-2008, New York*, 2008.

[19] W. Kahan and J. D. Darcy. How Javas floating-point hurts everyone everywhere. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998. `http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf`

[20] T. Kauranne, J. Oinonen, S. Saarinen, O. Serimaa, and J. Hietaniemi. The operational HIRLAM 2 on parallel computers. In *6th Workshop on the use of parallel processors in meteorology, ECMWF*, pages 63–74, November 1994.

[21] B. Krammer, K. Bidmon, M.S. Muller, and M.M. Resch. MARMOT: An MPI analysis and checking tool. *Advances in Parallel Computing*, 13:493–500, 2004.

[22] C. Lauter and V. Ménissier-Morain. There's no reliable computing without reliable access to rounding modes. In *SCAN 2012 Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics*, Russian Federation, 99–100, 2012

[23] M. Leeser, J. Ramachandran, T. Wahl, and D. Yablonski. OpenCL Floating Point Software on Heterogeneous Architectures – Portable or Not? In *Workshop on Numerical Software Verification (NSV)*, 2012. `http://www.cs.rice.edu/~sc40/NSV/12/program.html`

[24] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, A. Kapur, M. Martin, T. Tung, and D.J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software*, 28(2):152–205, 2002.

[25] E. McIntosh, F. Schmidt, and F. de Dinechin. Massive Tracking on Heterogeneous Platforms. In *ICAP 2006*, Chamonix, France, 2006.

[26] B. Marker, F. Van Zee, K. Goto, G. Quintana-Orti, and R. van de Geijn. Toward scalable matrix multiply on multithreaded architectures. *Euro-Par 2007 Parallel Processing*, pages 748–757, 2007.

[27] J. K. Martinsen and H. Grahn. A Methodology for Evaluating JavaScript Execution Behavior in Interactive Web Applications. In *AICCSA 2011, 9th IEEE/ACS International Conference on Computer Systems and Applications*, pages 241–248, December 2011. ‘

[28] R. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, 1966.

[29] R. Moore, R.B. Kearfott, and M.J. Cloud. *Introduction to interval analysis*. SIAM, 2009.

[30] J.-M. Muller et al. *Handbook of Floating-point Arithmetic*. Birkhäuser, 2010.

[31] A. Neumaier. *Interval methods for systems of equations*. Cambridge University Press, 1990.

[32] H. D. Nguyen. *Efficient algorithms for verified scientific computing : Numerical linear algebra using interval arithmetic*. Phd thesis, ENS de Lyon, January 2011. `http://tel.archives-ouvertes.fr/tel-00680352/PDF/NGUYEN_Hong_-_Diep_2011_These.pdf`

[33] M. Philippsen. Is Java ready for computational science? In *Euro-PDS'98 - 2nd European Parallel and Distributed Systems Conference*, pages 299–304, Austria, 1998.

[34] M. Philippsen, R. F. Boisvert, V. S. Getov, R. Pozo, J. Moreira, D. Gannon, and G. C. Fox. JavaGrande – High Performance Computing with Java. In *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*, pages 20–36. Springer, 2001.

[35] N. Revol, Y. Denneulin, J.-F. Méhaut, and B. Planquelle. Parallelization of continuous verified global optimization. In *19th IFIP TC7 Conf. on System Modelling and Optimization, Cambridge, United Kingdom*, 1999.

[36] N. Revol, K. Makino, and M. Berz. Taylor models and floating-point arithmetic: proof that arithmetic operations are validated in COSY. *Journal of Logic and Algebraic Programming*, 64:135–154, 2005.

[37] N. Revol, and P. Théveny. Parallel implementation of interval matrix multiplication. `http://hal.inria.fr/hal-00801890`, 2013.

[38] N. Revol. Interval Matrix Multiplication Oblivious to the Rounding Mode. in preparation, 2013.

[39] R. W. Robey, J. M. Robey, and R. Aulwes. In search of numerical consistency in parallel programming. *Parallel Computing*, 37:217–229, 2011.

[40] S. M. Rump. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77-104. Kluwer Academic Publishers, Dordrecht, 1999.

[41] S. M. Rump. Fast interval matrix multiplication. *Numerical Algorithms*, 61(1):1–34, 2012.

[42] V. Stodden, D. H. Bailey, J. Borwein, R. J. LeVeque, W. Rider, and W. Stein. Setting the Default to Reproducible – Reproducibility in Computational and Experimental Mathematics. Report on ICERM workshop *Reproducibility in Computational and Experimental Mathematics*, December 2012. ICERM, Brown University, Providence, RI, USA, 2013. `http://stodden.net/icerm_report.pdf`

[43] M. Taufer, O. Padron, P. Saponaro, and S. Patel. Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–9, Atlanta, GA, USA, April 2010.

[44] G. K. Thiruvathukal. Java at Middle Age: Enabling Java for Computational Science. *Computing in Science & Engineering*, 2(1):74–84, 2002.

[45] F. Tisseur. Newton's method in floating point arithmetic and iterative refinement of generalized eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications*, 22(4):1038–1057, 2001.

[46] R. Todd. Introduction to Conditional Numerical Reproducibility (CNR), 2012. `http://software.intel.com/en-us/articles/introduction-to-the-conditional-numerical-reproducibility-cnr`

[47] W. Tucker. *Validated Numerics: A Short Introduction to Rigorous Computations*. Princeton University Press, 2011.

[48] O. Villa, D. Chavarrí a Miranda, V. Gurumoorthi, A. Màrquez, and S. Krishnamoorthy. Effects of Floating-Point non-Associativity on Numerical Computations on Massively Multithreaded Systems. In *Cray User Group*, Atlanta, GA, USA, 2009. `https://cug.org/5-publications/proceedings_attendee_lists/CUG09CD/S09_Proceedings/pages/authors/01-5Monday/4C-Villa/villa-paper.pdf`