ESAIM: PROCEEDINGS AND SURVEYS, September 2014, Vol. 45, p. 229-238
J.-S. Dhersin, Editor

FIRST STEPS TOWARDS MORE NUMERICAL REPRODUCIBILITY *

Fabienne Jézéquel[1], Philippe Langlois[2] and Nathalie Revol[3]

Abstract. Questions whether numerical simulation is reproducible or not have been reported in several sensitive applications. Numerical reproducibility failure mainly comes from the finite precision of computer arithmetic. Results of floating-point computation depends on the computer arithmetic precision and on the order of arithmetic operations. Massive parallel HPC which merges, for instance, many-core CPU and GPU, clearly modifies these two parameters even from run to run on a given computing platform. How to trust such computed results? This paper presents how three classic approaches in computer arithmetic may provide some first steps towards more numerical reproducibility.

1. Numerical reproducibility: context and motivations

As computing power increases towards exascale, more complex and larger scale numerical simulations are performed in various domains. Questions whether such simulated results are reproducible or not have been reported more or less recently, *e.g.* in energy science [1], dynamic weather forecasting [2], atomic or molecular dynamic [3,4], fluid dynamic [5]. This paper focuses on numerical non-reproducibility due to the finite precision of computer arithmetic – see [6] for other issues about "reproducible research" in computational mathematics.

The following example illustrates a typical failure of numerical reproducibility. In the energetic field, power system state simulation aims to compute at "real time" a reliable estimate of the bus voltages for a given power grid topology and a set of on-line measures. Numerically speaking, a large and sparse linear system is solved at every iteration of a Newton-Raphson process. The core computation is a sparse matrix-vector product automatically parallelised by the computing environment. The authors of [1] exhibit a significant variability (up to 25% relative difference) between two runs on a massively multithreaded system. The culprit? Here as in the previously cited references: non deterministic sums.

Floating-point summation is not associative. Parallelism introduces non deterministic events from run to run, even using a unique binary file and a given computing platform. The order of communications, the number of computing units (threads, processors) and the associated data placement may vary, and hence the parallel partial sums. Even sequential frames that comply with the IEEE-754 floating-point arithmetic standard [7] are still numerically very sensitive to many features: low-level arithmetic unit properties (variable precision registers, fused operators), compiler optimizations, language flaws or library versions reduce numerical repeatability and numerical portability [8].

* *Authors thank Cl.-P. Jeannerod (INRIA) for his significant contribution, I. Said (LIP6) for his help in the numerical experiments related to the acoustic wave equation and the GT Arith, GDR Informatique Mathématique, for its support.*

[1] Sorbonne Universités, UPMC Univ Paris 06, CNRS, UMR 7606, LIP6, F-75005, Paris, France and Université Paris 2, France. Fabienne.Jezequel@lip6.fr

[2] Université de Perpignan Via Domitia, DALI, LIRMM (UMR5506 CNRS-Université Montpellier 2), France. langlois@univ-perp.fr

[3] INRIA, U. de Lyon, LIP (UMR5668 CNRS-ENS de Lyon-INRIA-UCBL), ENS de Lyon, France. Nathalie.Revol@ens-lyon.fr

Article published online by EDP Sciences and available at http://www.esaim-proc.org or http://dx.doi.org/10.1051/proc/201445023

Causes for such failures are numerous. So are the levels of an expected numerical reproducibility: from bit-wise identical results to debug or to verify contractual specifications, to a limited but guaranteed result accuracy to validate large scale simulations. Nevertheless numerical reproducibility failure is sometimes welcome as it may reveal a hidden bug. Of course numerical quality and running time or memory efficiency are contradictory goals that yield very different implementations in most cases.

Most of these failure causes are well known. Existing literature is not rare but spread over the concerned scientific communities: computer arithmetic, parallelism and HPC or over the numerous application domains [9]. Nevertheless the recent publications we mentioned at the beginning illustrate that the efficient control of their effects is still an open problem as exascale HPC allows application domain experts to trustfully simulate more and more realistic models. The aim of this paper is to present how three computer classic approaches in computer arithmetic may provide different response levels towards more numerical reproducibility.

Correct rounding, *i.e.* infinite precision rounded to a target format, ensures numerical reproducibility. An efficient and correctly rounded algorithm is very difficult to design and even more *to prove*. Section 2 illustrates how to go further than the classic backward error analysis for small numerical kernels. Here an algorithm specifically designed to compute 2 by 2 determinants with IEEE-754 arithmetic is proved to always return an highly accurate result. Computing a set that encloses the infinite precision solution provides a safe reference *to validate* non-reproducible computed results. Interval arithmetic is well known to return such set but, in practice, is implemented with floating-point arithmetic. Section 3 discusses how interval arithmetic faces the numerical reproducibility failures of its implementations and exhibits how to circumvent one important risk: the loss of the inclusion property when the rounding modes vary with respect to the computing platforms. Efficient validation with interval arithmetic needs more than often *ad hoc* algorithms. Section 4 presents a more general probabilistic approach *to estimate* the local accuracy lost and so to localise some sources of numerical non-reproducibility. Discrete stochastic arithmetic implemented in the CADNA library is applied to a 3D acoustic wave problem that suffers from different numerical reproducibility failures across CPU, GPU and APU.

Simulating an arbitrarily precise arithmetic is always possible: for instance double-double or quad-double libraries yield about twice or four times the IEEE-754 binary64 precision [10]. Most of the introductory references apply these libraries to improve numerical reproducibility without however avoiding all failure cases. Whereas they provide the first steps towards correct rounding, these libraries are not considered hereafter; the reader is invited to study [11] for details.

2. Proving tight a priori bounds on round-off errors

Floating-point arithmetic as specified by the IEEE-754 standard is the most common way of dealing with real numbers on a computer. In numerical analysis, floating-point arithmetic is most often described in a simplified manner by the so-called *standard model*, which for each of the basic operations $+$, $-$, $\times$, $/$, $\sqrt{}$ provides an upper bound on the rounding error possibly incurred by that operation. This model has been successfully used for more than 50 years to analyze a priori the numerical behavior of many algorithms, and especially linear algebra algorithms; see for example [12].

In particular, this model makes it possible to derive (in an efficient and elegant manner) some stability results and error bounds that are valid independently of the implementation of basic building blocks like sums of n floating-point numbers or dot products of two n-dimensional vectors. In other words, such *a priori* error bounds can be seen as a first step towards numerical reproducibility: depending on the context of computation (ordering, etc.), different results can (and, most of the time, will) be obtained, but we have at least the guarantee that they all share a same, common error bound.

On the other hand, many "low-level" features of IEEE-754 arithmetic are of course not captured by the standard model. This is most probably just the price to pay for having a model that is simple enough to be practical, but this sometimes leads to unnecessarily weak or pessimistic conclusions. As we shall see, this is already the case for some small numerical kernels like 2 by 2 determinants. We will also see that, in contrast,

exploiting "low-level" features in such cases makes it possible to derive *a priori* relative error bounds that are always tiny (that is, equal to a small integer multiple of the unit roundoff) and, therefore, to predict that for such kernels high relative accuracy is in fact always ensured.

2.1. **A reminder on IEEE floating-point arithmetic**

The most recent specification of floating-point arithmetic is the 2008 revision of the IEEE-754 standard [7]. This standard, which has aimed since its initial 1985 version at better robustness, efficiency, and portability of numerical programs, specifies in particular the following issues.

- Floating-point data and their encoding into integers for various formats;
- Results of operations on such data, as well as rounding-direction attributes;
- Exceptional behaviors and how they should be handled;
- Conversions between formats.

Now we briefly comment on several of the above aspects; for thorough treatments, we refer to [8].

The data specified by IEEE-754-2008 consist of finite nonzero floating-point numbers and of special data. Finite nonzero floating-point numbers have the form

$$x = (-1)^s \cdot m \cdot \beta^e, \tag{1}$$

where $s \in \{0,1\}$ is the sign bit, $m = \sum_{i=0}^{p-1} m_i \beta^{-i}$, $m_i \in \{0,1,\ldots,\beta-1\}$, is the significand, and $e \in \{e_{\min},\ldots,e_{\max}\}$ is the exponent. Here, the integers $\beta, p, e_{\min}, e_{\max}$ are the parameters defining the format of such numbers: for example, the well-known binary64 format (double-precision) has radix $\beta = 2$, precision $p = 53$, and extremal exponents $e_{\min} = -1022$ and $e_{\max} = 1023$. Furthermore, the number x is said to be normal when $m_0 \neq 0$, and subnormal when $m_0 = 0$ and $e = e_{\min}$. Special data consist of signed zeros $+0$ and -0, infinities $+\infty$ and $-\infty$, as well as not-a-numbers (NaN).

The result of operations like $+$, $-$, $\times$, $/$, $\sqrt{}$ must obey the so-called *correct rounding* rule, which says that the operation is performed "as if to infinite precision" and then rounded to the target format. This requires to define how to round real numbers to floating-point data. IEEE-754 specifies five ways of doing so, via rounding-direction attributes: round to nearest (RN) with two different tie-breaking strategies, round towards zero (RZ), round towards $+\infty$ (RU), and round towards $-\infty$ (RD). The default way of breaking ties is "to even", the other one being "to away"; see [7, §4.3.1]. Note also that even if round to nearest tends to be the default rounding mode in practice, the directed roundings RD and RU are essential for interval arithmetic; see Section 3.

Two specific features of the 2008 revision of IEEE-754 are worth being mentioned. First, the *fused multiply-add* (FMA) operation is now fully specified: given floating-point numbers a, b, c, this operation evaluates the expression $ab + c$ with a single rounding instead of two. Second, correct rounding is recommended — yet, not required, for so-called elementary functions, which include functions like exp, cos, x^n, $\sqrt{x^2+y^2}$, and so on.

Finally, special data are a way of coping with exceptional behaviors. For example, if one of the operands of an operation is a NaN, or in cases like $0/0$ or $\sqrt{-1}$, then the operation should be considered invalid and a NaN should be returned. Similarly, when the exact result of an operation lies outside the normal floating-point range, then either gradual underflow occurs and a subnormal number is returned, or overflow occurs and then $\pm\infty$ or the smallest/largest representable number is returned.

2.2. **The standard model of floating-point arithmetic**

The standard model is a simple and effective way of expressing the main feature of IEEE floating-point arithmetic: thanks to correct rounding and unless an exceptional behavior occurs, the result of any of the five basic arithmetic operations equals the exact result times a quantity close to 1. More precisely, given two normal floating-point numbers x and y, and given $\mathrm{op} \in \{+,-,\times,/\}$, there exists ϵ in $\mathbb{R}$ such that

$$\mathrm{RN}(x \,\mathrm{op}\, y) = (x \,\mathrm{op}\, y)(1+\epsilon), \qquad |\epsilon| \leqslant u := \frac{1}{2}\beta^{1-p}. \tag{2}$$

Following [12, p. 40] we shall refer to (2) as the *standard model of floating-point arithmetic*. The same kind of equality holds for square root and the FMA. Here, the number u is the *unit roundoff* associated with the floating-point format under consideration. For the usual formats, it is a tiny value and thus $1+\epsilon \approx 1$. Note also we have assumed in (2) that rounding is to nearest (RN). However, a similar model can be stated for directed roundings RZ, RD, RU, up to replacing u by $2u$.

A key feature of (2) is to make it possible to consider the computed result as being the exact result of the same operation on slightly perturbed data. For example, $\mathrm{RN}(x+y)$ equals the exact sum $\widetilde{x}+\widetilde{y}$, where $\widetilde{x}=x(1+\epsilon)$ and $\widetilde{y}=y(1+\epsilon)$ are real numbers whose relative distance to x and y, respectively, is bounded by u. Using this feature repeatedly is at the heart of Wilkinson's *backward error analysis* [13, 14]. As an example, consider the evaluation (with + and × and in any order) of the dot product $r=\sum_{i=1}^{n} x_i y_i$. Assuming that underflows and overflows do not occur, we can apply (2) to each of the $2n-1$ operations involved and deduce that the computed solution satisfies $\widehat{r}=\sum_{i=1}^{n} x_i y_i(1+\epsilon_i)$, where for $i=1,\ldots,n$, $|\epsilon_i| \leqslant (1+u)^n-1 = nu+O(u^2)$. This equality says $\widehat{r}$ is the exact dot product of slight real perturbations of the initial floating-point vectors. For r nonzero, it leads further to the forward relative error bound

$$\frac{|\widehat{r}-r|}{|r|} \leqslant \big((1+u)^n-1\big)\frac{\sum_{i=1}^{n}|x_i y_i|}{|r|}. \tag{3}$$

When $\sum_{i=1}^{n}|x_i y_i| \gg |r|$, such a bound can be larger than 1. However, this agrees with $\widehat{r}$ having no correct digit at all, because of possible catastrophic cancellations occurring when adding the n products.

Let us now consider a second example, namely, Kahan's algorithm for 2 by 2 determinants [12, p. 60], where just using the standard model (2) is *not* enough to obtain a satisfactory error bound. Given floating-point numbers a, b, c, d and assuming that an FMA is available, Kahan's algorithm evaluates $r=ad-bc$ as follows:

$$\widehat{w} := \mathrm{RN}(bc); \qquad \widehat{f} := \mathrm{RN}(ad-\widehat{w}); \qquad e := \mathrm{RN}(\widehat{w}-bc); \qquad \widehat{r} := \mathrm{RN}(\widehat{f}+e).$$

The operation producing e being exact [15], it can be checked by applying the standard model (2) to each of the 3 remaining operations that the computed result $\widehat{r}$ satisfies

$$\frac{|\widehat{r}-r|}{|r|} \leqslant (2u+u^2)\left(1+\frac{u|bc|}{2|r|}\right). \tag{4}$$

Hence this bound implies high relative accuracy as long as $u|bc| \not\gg 2|r|$, and when the ratio $\frac{u|bc|}{2|r|}$ is huge, one could believe that the error itself can be very large too. In fact, as shown in [16], the latter situation turns out to be impossible. However to prove Kahan's algorithm is always highly accurate, it is not enough to use just the standard model (2) and the exactness of e: further properties of floating-point arithmetic must be exploited.

2.3. **Exploiting "low-level" features of IEEE floating-point arithmetic**

In addition to (2), the IEEE-754 specification confers many nice properties to floating-point arithmetic, thanks to the special shape (1) of floating-point numbers and to the fact that rounding $r \in \mathbb{R}$ to nearest satisfies, by definition, $|\mathrm{RN}(r)-r| \leqslant |x-r|$ for all floating-point numbers x. Such properties include the fact that if $x \operatorname{op} y$ is a floating-point number then $\epsilon=0$, the fact that RN commutes with the absolute value and is nondecreasing, or the fact that floating-point multiplication by an integer power of the radix β is exact. Some other properties heavily depend on the notion of *unit in the last place* (ulp) [8, §2.6.1]: in particular, given $r \in \mathbb{R}$ and in the absence of underflow and overflow, the following tighter bound implies the standard model (2):

$$|\mathrm{RN}(r)-r| \leqslant \mathrm{ulp}(r)/2 \leqslant u|r|.$$

In Kahan's algorithm, the error terms $\epsilon_1 := \widehat{f}-(ad-\widehat{w})$ and $\epsilon_2 := \widehat{r}-(\widehat{f}+e)$ can be analyzed tightly thanks to those "lower-level" properties: as proved in [16], both $|\epsilon_1|$ and $|\epsilon_2|$ are bounded by $\frac{\beta}{2}\mathrm{ulp}(r)$ and, on the other hand, $\widehat{r}-r=\epsilon_1+\epsilon_2$. Consequently, the relative error of Kahan's algorithm is bounded as $|\widehat{r}-r| \leqslant 2\beta u|r|$. Since

in practice β is 2 or 10, the bound $2\beta u$ thus already ensures that Kahan's algorithm always has high relative accuracy, in contrast to (4). However, this bound is still sub-optimal and it is shown further in [16] that

$$\frac{|\widehat{r} - r|}{|r|} \leqslant 2u$$

for all radix $\beta \geqslant 2$, and that this bound is tight. Thus, a careful use of ulp properties and of various other "lower-level" features of IEEE-754 arithmetic made it possible to prove that a numerical kernel like Kahan's 2 by 2 determinant always behaves ideally in the absence of underflow and overflow. This was far from obvious when resorting exclusively to the standard model.

3. Validating numerical results with interval arithmetic

Interval arithmetic is a tool of choice when it comes to guaranteeing the result of a numerical computation. Indeed, interval computations return intervals which are enclosures of the exact results, if the inputs are intervals enclosing the data. This holds true even when floating-point arithmetic is used and round-off errors enter the scene, and even when numerical computations are not performed reproducibly. When the numerical computation is not reproducible, that is, when several runs produce different results, these results can be compared against the interval result.

3.1. **Introduction to interval arithmetic**

Interval computations handle intervals, not numbers. An interval can be the enclosure of some physical data d, measured with, say, absolute error at most δ: in this case the corresponding interval is $[d-\delta, d+\delta]$. An interval can also be used to process a value that is not finitely representable, such as π: the interval $[3.1415, 3.1416]$ contains π, it is finitely representable in decimal arithmetic and it can easily be manipulated. Finally, intervals can be used to handle whole ranges of values, to compute with sets.

Computations can be performed on intervals. An interval that appears as an intermediate or final quantity in a computation is an enclosure of the range of possible values for the corresponding numerical quantity. Let us put it more explicitly. In interval arithmetic, the fundamental theorem is the *inclusion property: the exact result, be it a number or a set, is contained in the computed interval.* It is guaranteed that no result is lost, *i.e.* lies outside the interval.

If the input data satisfy the inclusion property, it is possible to perform subsequent operations and computations in a way that preserves the inclusion property. Let us give two examples, the addition and the division. In what follows, let us denote intervals using boldface letters and let us represent them by their endpoints. An interval $\mathbf{a}$ is a subset of $\mathbb{R}$ and $\mathbf{a} = [\underline{a}, \bar{a}] = \{a : \underline{a} \leq a \leq \bar{a}\}$ with $\underline{a} \in \mathbb{R} \cup \{-\infty\}$, $\bar{a} \in \mathbb{R} \cup \{+\infty\}$ and $\underline{a} \leq \bar{a}$. Let $\mathbf{a} = [\underline{a}, \bar{a}]$ and $\mathbf{b} = [\underline{b}, \bar{b}]$ be two intervals. The addition $\mathbf{c} = \mathbf{a} + \mathbf{b}$ is obtained using the formula $\mathbf{c} = [\underline{c}, \bar{c}] = [\underline{a} + \underline{b}, \bar{a} + \bar{b}]$, *i.e.* $\underline{c} = \underline{a} + \underline{b}$ and $\bar{c} = \bar{a} + \bar{b}$. If $\mathbf{a} \not\ni 0$, then the inverse $1/\mathbf{a}$ is given by $1/\mathbf{a} = [1/\bar{a}, 1/\underline{a}]$. These examples illustrate that it is possible to compute with intervals. The general definition of $\mathbf{a} \diamond \mathbf{b}$, where $\diamond$ is any mathematical operation, is

$$\mathbf{a} \diamond \mathbf{b} = \mathrm{Hull}\{a \diamond b,\ a \in \mathbf{a},\ b \in \mathbf{b}\},$$

where Hull is the convex hull of the set: this ensures that the result is an interval, without gaps. As it can be seen on the formulas for addition and inversion, using the monotonicity leads to practical formulas for the implementation of operations. For more details about interval arithmetic, see [17, 18] for instance.

Frequently, interval arithmetic is implemented on computers using the floating-point arithmetic available on these machines. To preserve the inclusion property, directed roundings (introduced in Section 2.1) of the endpoints are used. For instance, the formulas for the addition and the division become

$$\mathbf{a} + \mathbf{b} = [\mathrm{RD}(\underline{a} + \underline{b}), \mathrm{RU}(\bar{a} + \bar{b})], \quad 1/\mathbf{a} = [\mathrm{RD}(1.0/\bar{a}), \mathrm{RU}(1.0/\underline{a})],$$

As already mentioned, the main advantage of interval arithmetic is to offer a guarantee: the computed intervals are guaranteed to contain the exact results. It is also said that interval arithmetic returns *verified enclosures* of the results. It has been hoped, in the early years of interval arithmetic, that it would in particular provide a guarantee on the numerical accuracy of a result computed using floating-point arithmetic, *i.e.* that it would give the error due to round-off. Unfortunately, this is not the case: replacing, in a numerical code, the floating-point datatype by the interval datatype may produce very large intervals. Indeed, interval computations suffer from overestimation and such a naive use of interval arithmetic is very likely to yield to a blow-up of the interval widths. Let us illustrate the problem with the simplest example possible. Computing $\mathbf{a} - \mathbf{a}$ does not result in $\{0\}$, as with real or floating-point arithmetic, but in an interval whose width is twice the width of $\mathbf{a}$. Let us use the definition of an operation to understand the problem:

$$\mathbf{a} - \mathbf{a} = \text{Hull}\{a_1 - a_2,\ a_1 \in \mathbf{a},\ a_2 \in \mathbf{a}\} = [\underline{a} - \bar{a}, \bar{a} - \underline{a}].$$

What this example illustrates is the so-called *variable dependency* problem: it is not known whether a_1 and a_2 correspond to the same variable or not, and thus to the same value in $\mathbf{a}$ or not. This example is not contrived: Karatsuba's formula for the multiplication of polynomials is $(a_1 x + a_0) \cdot (b_1 x + b_0) = a_1 b_1 x^2 + ((a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0)\, x + a_0 b_0$, and, therefore, exploits the fact that $a_1 b_1 - a_1 b_1 = a_0 b_0 - a_0 b_0 = 0$, which does not hold true in interval arithmetic. Strassen's algorithm for fast matrix multiplication is similar.

Two main families of techniques are commonly used to reduce this interval's swelling. A brute force approach, which is often used as a last resort, consists in bisecting the input intervals in smaller subintervals, and in performing the computations on these subintervals. The final result is the union of the sub-results. It is known [17, §2.1] that the width of the result is proportional to the width of the input: this justifies the bisection technique. Another family of techniques consists in employing more elaborate variants of interval arithmetic, designed to reduce the dependency problem. Let us cite briefly slope and other so-called centered forms [17, §2.3], affine arithmetic [19,20], Taylor models [21] among the most used ones. Algorithms are designed specifically for interval computations and most of them usually combine both kinds of techniques.

3.2. **Interval arithmetic and reproducibility issues: when the inclusion property is at risk**

As interval arithmetic is usually implemented using floating-point arithmetic, it is subject to the same issues, regarding numerical reproducibility, as floating-point arithmetic. The main issues, as introduced in Section 1, are variable computing precision and variable order of the operations. It is also subject to specific issues, in particular the order, not only of the arithmetic operations but also of the bisections of intervals. Interval arithmetic is sensitive to numerical reproducibility issues: the computed result may differ from one run to the next. Still, it is always guaranteed to be an enclosure of the result. However, in some cases, this guarantee can be lost. Indeed, other issues regarding numerical reproducibility of interval computations threaten the inclusion property.

The main threat is related to the use of several rounding modes. As illustrated in Section 3.1, the implementation of interval arithmetic using floating-point arithmetic relies heavily on the use of the directed rounding modes RD and RU. This point makes the main difference between numerical reproducibility for floating-point computations (which use rounding to nearest) and for interval ones. Indeed, rounding modes set by the user or by the interval library are sometimes modified. Compiler optimization may be aggressive to gain performance and may exchange the relative order of rounding mode settings and of arithmetic operations, thus modifying incorrectly the computed result (see for example the gcc bug report ♯34678). The programming language may not support changes of the rounding mode: OpenCL and OpenMP only support rounding-to-nearest. The execution environment for multithreaded computations may or may not handle properly rounding modes that are attached to different threads. For some architectures such as GPUs with CUDA, rounding modes are given in the code of the operations and are accessible via the programming language, thus they are properly and smoothly handled. For most other architectures this is unfortunately not the case. Finally, external numerical libraries may have their own handling of rounding modes. At each step, how rounding modes are handled is usually poorly documented or not documented at all.

3.3. **How to preserve the inclusion property**

The inclusion property can be lost due to the unavailability of directed rounding modes. The inclusion property can therefore be recovered by becoming independent of the rounding mode. Indeed, interval arithmetic can be implemented with the default rounding to nearest. This can be done by enlarging the computed interval, as in the early version of the FI_LIB library [22] for instance: for each operation, the last bit of the left endpoint was decreased by 1, and similarly the right endpoint was slightly shifted right. Some experiments indicate that the loss of accuracy is limited: it roughly corresponds to computing with a precision of 52 bits instead of 53 in binary64. Another approach, promoted in [23] and here, consists in determining bounds on the round-off errors, such as bounds given in (3) for the dot product, and by shifting the endpoints by this amount. However, the bound given in (3) uses real arithmetic, whereas we need an upper bound that can be computed with floating-point arithmetic. Let us take the example of the dot product: the idea is to enlarge every operand $|x_i|$ and $|y_i|$ by a multiplicative factor — and we suggest to use the factor $1+4u$, which is a floating-point number —, and to perform a floating-point multiplication. Subsequently, the floating-point product $\mathrm{RN}(\mathrm{RN}((1+4u)\cdot|x_i|)\cdot\mathrm{RN}((1+4u)\cdot|y_i|))$ is an upper bound of $|x_iy_i|$. Eventually the sum, in any order, of these products is an upper bound of $\sum_{i=1}^{n}|x_iy_i|$. Similarly, a floating-point multiplication by a term slightly larger than $\left((1+u)^n-1\right)$ ensures that an upper bound on the right hand-side of (3) has been obtained, by resorting solely to floating-point arithmetic. It can even be obtained by using any optimized numerical routine for the dot product, where the input vectors are $(\mathrm{RN}((1+4u)\cdot|x_i|))_{1\le i\le n}$ and $(\mathrm{RN}((1+4u)\cdot|y_i|))_{1\le i\le n}$. A similar method is given in [23] for the product of matrices. What this dot product exemplifies is the fact that the inclusion property can be preserved even when the only available mode is rounding to nearest (RN), and that interval computations can even benefit from optimized numerical routines. The only condition is that these bounds must hold independently of issues related to numerical reproducibility: in particular, they have to hold independently of the order of the operations.

4. Estimating round-off errors with discrete stochastic arithmetic

Differences between the results of a code can be observed from one architecture to another or even inside the same architecture. In this section, we show how to estimate round-off errors which lead to such differences. The aim here is not to force a code to be reproducible, but to estimate the number of digits in the results which may be different from one execution to another because of round-off errors. This approach is illustrated by a wave propagation code which can be affected by reproducibility problems.

4.1. **Reproductibility failures in a wave propagation code**

We consider the three-dimensional acoustic wave equation $\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}-\sum_{b\in\{x,y,z\}}\frac{\partial^2}{\partial b^2}u=0$, where u is the particle velocity, c is the wave velocity, and t is the time. This equation, used for instance in oil exploration [24], is solved with a finite difference scheme of order 2 in time and p in space (in our case $p=8$).

Two mathematically equivalent implementations of the finite difference scheme are proposed:

$$u_{ijk}^{n+1}=2u_{ijk}^{n}-u_{ijk}^{n-1}+\frac{c^2\Delta t^2}{\Delta h^2}\sum_{l=-p/2}^{p/2}a_l\left(u_{i+ljk}^{n}+u_{ij+lk}^{n}+u_{ijk+l}^{n}\right)+c^2\Delta t^2 f_{ijk}^{n}, \tag{5}$$

$$u_{ijk}^{n+1}=2u_{ijk}^{n}-u_{ijk}^{n-1}+\frac{c^2\Delta t^2}{\Delta h^2}\left(\sum_{l=-p/2}^{p/2}a_l u_{i+ljk}^{n}+\sum_{l=-p/2}^{p/2}a_l u_{ij+lk}^{n}+\sum_{l=-p/2}^{p/2}a_l u_{ijk+l}^{n}\right)+c^2\Delta t^2 f_{ijk}^{n}, \tag{6}$$

where u^n_{ijk} (respectively f^n_{ik}) is the wave (respectively source) field in (i, j, k) coordinates and nth time step and a_l, $l = -p/2, \ldots, p/2$, are the finite difference coefficients.

The code is executed for $64 \times 64 \times 64$ space steps and 1000 time iterations in IEEE-754 binary32 arithmetic with rounding to the nearest and the following CPU, GPU and APU (Accelerated Processing Unit): i) Intel i7-3687U CPU with gcc 4.6.3 compiler; ii) NVIDIA K20c GPU (Graphics Processing Unit) with OpenCL (Open Computing Language); iii) AMD Radeon HD 7970 GPU with OpenCL; iv) AMD Trinity APU with OpenCL.

Different kinds of reproducibility problems are observed. The results numerically vary

(1) from one execution to another inside a GPU or an APU; these repeatability problems are due to differences in the execution order of the threads;
(2) from one implementation of the finite difference scheme to another; the maximal relative difference between results is of the order of 10^{-1} to 1 depending on the architecture, and the mean value of the relative difference between results is of the order of 10^{-5} whatever the architecture;
(3) from one architecture to another; again, the maximal relative difference between results is of the order of 10^{-1} to 1 and its mean value is 10^{-5}.

Therefore, if two sets of results computed in binary32 are compared, the results at the same space coordinates can have from 0 to 7 significant digits in common, and the average number of common significant digits is about 4. We recall that the binary32 arithmetic precision is about 8 digits.

4.2. **Principles of discrete stochastic arithmetic (DSA)**

Based on a probabilistic approach, the CESTAC method [25] allows the estimation of round-off error propagation which occurs with floating-point arithmetic. When no overflow occurs, the exact result of any non exact floating-point arithmetic operation is bounded by two consecutive floating-point values R^- and R^+. The basic idea of the method is to perform each arithmetic operation N times, randomly rounding each time, with a probability of 0.5, to R^- or R^+. The computer's deterministic arithmetic, therefore, is replaced by a stochastic arithmetic where each arithmetic operation is performed N times before the next one is executed, thereby propagating the round-off error differently each time. The CESTAC method furnishes us with N samples $R_1, \ldots, R_N$ of the computed result R. The value of the computed result, $\overline{R}$, is the mean value of $\{R_i\}_{1 \leqslant i \leqslant N}$ and the number of exact significant digits in $\overline{R}$, $C_{\overline{R}}$, is estimated as

$$C_{\overline{R}} = \log_{10}\left(\frac{\sqrt{N}\,|\overline{R}|}{\sigma\tau_\beta}\right), \quad \text{where } \overline{\mathrm{R}} = \frac{1}{\mathrm{N}}\sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{R_i} \text{ and } \sigma^2 = \frac{1}{\mathrm{N}-1}\sum_{\mathrm{i}=1}^{\mathrm{N}} \left(\mathrm{R_i} - \overline{\mathrm{R}}\right)^2. \tag{7}$$

τ_β is the value of Student's distribution for $N - 1$ degrees of freedom and a probability level $1 - \beta$. In practice $N = 3$, $\beta = 0.05$ and then $\tau_\beta = 4.303$.

The validity of $C_{\overline{R}}$ is compromised if the two operands in a multiplication or the divisor in a division are not significant [26]. It is essential, therefore, that any computed result with no significance is detected and reported, since its subsequent use may invalidate the method. The need for this dynamic control of multiplications and divisions has led to the concept of the computational zero. A computed result is a computational zero, denoted by @.0, if $\forall i, R_i = 0$ or $C_{\overline{R}} \leq 0$. This means that a computational zero is either a mathematical zero or a number without any significance, *i.e.* numerical noise.

To establish consistency between the arithmetic operators and the relational operators, discrete stochastic relations are defined as follows. Let X and Y be two computed results, we have from [27]

$$X = Y \text{ if } \mathrm{X} - \mathrm{Y} = @.0; \quad \mathrm{X} > \mathrm{Y} \text{ if } \overline{\mathrm{X}} > \overline{\mathrm{Y}} \text{ and } \mathrm{X} - \mathrm{Y} \neq @.0; \quad \mathrm{X} \geq \mathrm{Y} \text{ if } \overline{\mathrm{X}} \geq \overline{\mathrm{Y}} \text{ or } \mathrm{X} - \mathrm{Y} = @.0. \tag{8}$$

Discrete Stochastic Arithmetic (DSA) is the combination of the CESTAC method, the concept of the computational zero, and the discrete stochastic relationships.

TABLE 1. Accuracy, number of occurrences, and range of the absolute values of the results.

Accuracy	#occurrences	absolute values
0	40	[5.1E–4, 1.2E–1]
1	414	[8.0E–3, 8.2E–1]
2	4,260	[1.4E–2, 1.4E+1]
3	39,364	[7.5E–2, 1.6E+2]
4	148,936	[8.3E–1, 8.3E+2]
5	67,130	[1.4E+0, 1.5E+3]
6	1,986	[1.3E+1, 1.2E+3]
7	14	[3.9E+1, 6.1E+2]

The CADNA software [28] is a library which implements DSA in any code written in C++ or in Fortran and allows to use new numerical types: the stochastic types. In practice, classic floating-point variables are replaced by the corresponding stochastic variables, which are composed of three perturbed floating-point values. The library contains the definition of all arithmetic operations and order relations for the stochastic types. The control of the accuracy is performed only on variables of stochastic type. Only significant digits of a stochastic variable are printed or "@.0" for a computational zero. Because all operators are overloaded, the use of CADNA in a program requires only a few modifications: essentially changes in the declarations of variables and in input/output statements. CADNA can detect numerical instabilities which occur during the execution of the code. When a numerical instability is detected, dedicated CADNA counters are incremented. At the end of the run, the value of these counters together with appropriate warning messages are printed on standard output.

4.3. **Estimation of numerical reproducibility by means of DSA**

The CPU version of the acoustic wave propagation code is examined with the CADNA library. With implementations (5) and (6) of the finite difference scheme, the number of exact significant digits in the results varies from 0 to 7, and its mean value is 3.6. This remark is consistent with the observations described in Section 4.1. Numerical instabilities are detected during the executions: 272,394 losses of accuracy due to cancellations with scheme (5) and 285,186 with scheme (6). These instabilities are detected if the subtraction of two close floating-point numbers leads to a loss of accuracy of at least four digits.

Table 1 presents the number of exact significant digits estimated by CADNA for scheme (5), the occurrence number of each accuracy, and the absolute value range of the analysed results. Similar results are observed with the other scheme. The highest results (in absolute value) are affected by low round-off errors and the highest round-off errors impact negligible results.

Results of numerical simulations may be different from one environment to another. Such differences are particularly noticeable with new computing architectures such as multicore processors, GPU and APU. Different orders in the sequence of floating-point operations lead to differences in round-off error propagation and therefore to reproducibility failures. DSA estimates which digits are affected by round-off errors in the simulation results. Therefore, it provides an estimation of the reproducibility of numerical programs. DSA enables, with the CADNA library, the numerical quality estimation of sequential programs in C or Fortran and of parallel programs using MPI for communication [29]. CADNA can also be used in hybrid CPU-GPU environments to estimate round-off errors in CUDA programs [30]. However, work must still be carried out to extend efficiently this approach to emerging computing architectures that are prone to numerical reproducibility failures.

References

[1] O. Villa, D. G. Chavarría-Miranda, V. Gurumoorthi, A. Márquez, and S. Krishnamoorthy. Effects of floating-point non-associativity on numerical computations on massively multithreaded systems. In *CUG 2009 Proceedings*, pages 1–11, 2009.

[2] Y. He and C.H.Q. Ding. Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications. *The Journal of Supercomputing*, 18:259–277, 2001.

[3] M. A. Cleveland, T. A. Brunner, N. A. Gentile, and J. A. Keasler. Obtaining identical results with double precision global accuracy on different numbers of processors in parallel particle Monte Carlo simulations. *Journal of Computational Physics*, 251(0):223 – 236, 2013.

[4] M. Taufer, O. Padron, Ph. Saponaro, and S. Patel. Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs. In *IPDPS*, pages 1–9. IEEE, 2010.

[5] R. W. Robey, J. M. Robey, and R. Aulwes. In search of numerical consistency in parallel programming. *Parallel Computing*, 37(4-5):217–229, 2011.

[6] 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. Technical report, ICERM, February 2013.

[7] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, August 2008.

[8] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[9] N. Revol and Ph. Théveny. Numerical reproducibility and parallel computations: Issues for interval algorithms. *IEEE Transactions on Computers*, 63(8):1915–1924, August 2014.

[10] URL = `http://crd.lbl.gov/~dhbailey/mpdist`, .

[11] D. H. Bailey, R. Barrio, and J. M. Borwein. High-precision computation: Mathematical physics and dynamics. *Applied Mathematics and Computation*, 218(20):10106–10121, 2012.

[12] N. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, (2nd edition), 2002.

[13] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Notes on Applied Science No. 32, Her Majesty's Stationery Office, London, 1963. Also published by Prentice-Hall, Englewood Cliffs, NJ, USA. Reprinted by Dover, New York, 1994.

[14] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, 1965.

[15] W. Kahan. Lecture notes on the status of IEEE Standard 754 for binary floating-point arithmetic. Manuscript, May 1996.

[16] C.-P. Jeannerod, N. Louvet, and J.-M. Muller. Further analysis of Kahan's algorithm for the accurate computation of 2×2 determinants. *Mathematics of Computation*, 82:2245–2264, 2013.

[17] A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, 1990.

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

[19] J. Stolfi and L. H. de Figueiredo. Self-validated numerical methods and applications. In *Monograph for 21st Brazilian Mathematics Colloquium, IMPA, Rio de Janeiro*, 1997.

[20] L. H. de Figueiredo and J. Stolfi. Affine arithmetic: concepts and applications. *Numerical Algorithms*, 37:147–158, 2004.

[21] K. Makino and Berz. M. Taylor models and other validated functional inclusion methods. *International Journal of Pure and Applied Mathematics*, 4(4):379–456, 2003.

[22] W. Hofschuster and W. Krämer. FI_LIB - A fast interval library (Version 1.2) in ANSI-C, 2005. `http://www2.math.uni-wuppertal.de/~xsc/software/filib.html`.

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

[24] H. Calandra, R. Dolbeau, P. Fortin, J.-L. Lamotte, and I. Said. Forward seismic modeling on AMD Accelerated Processing Unit. In *Rice Oil & Gas HPC Workshop, Houston, TX, USA*, March 2013.

[25] J. Vignes. A stochastic arithmetic for reliable scientific computation. *Math. Comput. Simulation*, 35:233–261, 1993.

[26] J.-M. Chesneaux. *L'arithmétique stochastique et le logiciel CADNA*. Habilitation à diriger des recherches, Université Pierre et Marie Curie, Paris, France, November 1995.

[27] J.-M. Chesneaux and J. Vignes. Les fondements de l'arithmétique stochastique. *C. R. Acad. Sci. Paris Sér. I Math.*, 315:1435–1440, 1992.

[28] F. Jézéquel and J.-M. Chesneaux. CADNA: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12):933–955, 2008.

[29] S. Montan and C. Denis. Numerical verification of industrial numerical codes. In *ESAIM*, volume 35, pages 107–113, 2012.

[30] F. Jézéquel and J.-L. Lamotte. Numerical validation of Slater integrals computation on GPU. In *14th International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN 2010)*, pages 78–79, Lyon, France, 2010.