Algorithm 1029: Encapsulated Error, a Direct Approach to Evaluate Floating-Point Accuracy

Nestor Demeure, Cédric Chevalier, Christophe Denis, Pierre Dossantos-Uzarralde

Algorithm 1029: Encapsulated Error, a Direct Approach to Evaluate Floating-Point Accuracy

NESTOR DEMEURE, Université Paris-Saclay, ENS Paris-Saclay, CNRS, Centre Borelli and CEA, DAM, DIF
CÉDRIC CHEVALIER, CEA, DAM, DIF
CHRISTOPHE DENIS, Sorbonne Université, CNRS, LIP6
PIERRE DOSSANTOS-UZARRALDE, CEA, DAM, DIF

Floating-point numbers represent only a subset of real numbers. As such, floating-point arithmetic introduces approximations that can compound and have a significant impact on numerical simulations. We introduce encapsulated error, a new way to estimate the numerical error of an application and provide a reference implementation, the Shaman library. Our method uses dedicated arithmetic over a type that encapsulates both the result the user would have had with the original computation and an approximation of its numerical error. We thus can measure the number of significant digits of any result or intermediate result in a simulation. We show that this approach, although simple, gives results competitive with state-of-the-art methods. It has a smaller overhead, and it is compatible with parallelism, making it suitable for the study of large-scale applications.

CCS Concepts: • **General and reference** → **Verification**; • **Mathematics of computing** → **Numerical analysis**; • **Software and its engineering** → **Dynamic analysis;**

Additional Key Words and Phrases: Floating-point arithmetic, numerical verification, round-off errors

ACM Reference format:
Nestor Demeure, Cédric Chevalier, Christophe Denis, and Pierre Dossantos-Uzarralde. 2023. Algorithm 1029: Encapsulated Error, a Direct Approach to Evaluate Floating-Point Accuracy. *ACM Trans. Math. Softw.* 48, 4, Article 47 (March 2023), 16 pages.
https://doi.org/10.1145/3549205

1 INTRODUCTION

Various kinds of errors can make a simulation diverge from the observed reality. Those include modeling errors, discretization errors, parameter uncertainties, as well as errors due to the use of floating-point arithmetic. We can now run more than 10^{15} floating-point operations per second on a supercomputer and 10^{12} floating-point operations per second on a standard GPU. However, as computing power increases, so does the size of simulations, the number of their arithmetic operations, and the magnitude of the values they process.

Authors' addresses: N. Demeure, Université Paris-Saclay, ENS Paris-Saclay, CNRS, Centre Borelli, 91190 Gif-sur-Yvette, France and CEA, DAM, DIF, F-91297 Arpajon, France; email: nestor.demeure@ens-paris-saclay.fr; C. Chevalier and P. Dossantos-Uzarralde, CEA, DAM, DIF, F-91297 Arpajon, France; emails: cedric.chevalier@cea.fr, pierre.dossantos-uzarralde@cea.fr; C. Denis, Sorbonne Université, CNRS, LIP6, 75005 Paris, France; email: christophe.denis@lip6.fr.

0098-3500/2023/03-ART47 $15.00
https://doi.org/10.1145/3549205

In this context, one can expect the rounding errors introduced by the use of floating-point arithmetic, also called *numerical errors* or *rounding errors*, to have an increasing impact on simulations. Their likelihood and their relative importance are increasing. This matters, as numerical errors can significantly degrade numerical results and introduce artifacts in physical simulations, and cause phenomena to be missed or misinterpreted [2].

In this article, we present a new way to estimate the numerical error of an application, which we call *encapsulated error*. We provide a method that is both direct (and thus easy to interpret) and accurate while keeping the overhead low, making it applicable to large computations.

We describe our method as direct because, by design, there is no proxy between the user and the numerical error. We compute the error locally and propagate it, encapsulating both the result of the original computation and an approximation of its numerical error in a dedicated type. Thus, we have direct access to the number of significant digits at all times and for all intermediate results. Moreover, having observed that some functions run on numbers with few actual significant digits, we can lower their precision to improve performance.

A C++ reference implementation, the Shaman library, is provided. It supports all mathematical functions in the current C++ standard library, as well as mixed-precision arithmetic, and is compatible with common parallelism frameworks (both OpenMP [7] and MPI [21]). Furthermore, it can be coupled with a traditional debugger such as GDB [45] to locate predefined types of numerical errors such as unstable tests.

Instrumenting classical algorithms, we illustrate how our method can be used to evaluate the quality of a result and show that it is accurate and fast. Its lower impact on computing times, compared with state of the art, and its compatibility with parallelism make it suitable for the study of large-scale applications.

The rest of the article is organized as follows. Section 2 is dedicated to a review of related work. Section 3 introduces the concept of numerical error and our working hypotheses. Section 4 explains our method. Section 5 details the theoretical properties of the method. Section 6 presents the reference implementation and associated tools. We illustrate the accuracy of the method and measure the overhead of our reference implementation in Section 7. We present our conclusion and perspectives in Section 8.

2 RELATED WORK

The quantification of the impact of floating-point arithmetic on a given computation, aiming not to correct it but to evaluate the result's numerical accuracy, is not new. A variety of methods have been developed, most of them deriving from a comparison with higher-precision arithmetic, static proof-based methods, interval arithmetic, or stochastic arithmetic.

The obvious solution is to compare, for each computation, their output with the result of an equivalent computation performed with higher-precision arithmetic (as explored in FpDebug [3], Precimonious [42], and Herbgrind [43]). Although simple to implement and interpret, this has two main limitations: first, high precision tends to have significant overhead (two orders of magnitude for MPFR [15] in our tests), and second, it relies on the hypothesis that the precision used will be sufficient to detect problems. An illustration is provided in Section 5.3, showing how a large cancelation (a subtraction between two numbers of similar magnitude) can make high-precision arithmetic unreliable when it is used to estimate the numerical accuracy of a result.

One can also use static methods, which are usually built on abstract interpretation (as in FLUCTUAT [18] and PRECiSA [14]), symbolic reasoning (as in FPTaylor [44]), SMT solvers (as in Rosa [10] and Daisy [8]), or proof assistants (as in Gappa [11] and Real2Float [34]) to prove that a result will have sufficient precision for all possible inputs. However, this approach is often

confined to programs written in domain-specific languages and tends to be limited to programs that are both short (often a single function) and simple (loops and tests increase the difficulty of the task considerably) [9].

Interval arithmetic [35, 36] (or one of its variants like affine arithmetic [6]) supplies a more scalable way to get strong guarantees on a result by doing the analysis dynamically, propagating not a number but an interval representing both a strict upper bound and a strict lower bound on the result and thus giving us an upper bound on the numerical error of the computation. In practice, when applied to large computations, those methods tend to return intervals too large to be useful, unless one performs highly specialized modifications of the original computation to mitigate the problem (Newton's method famously requires a fix to avoid diverging intervals [41]).

Stochastic arithmetic [39, 46] offers another alternative, adding noise to every arithmetic operation to study the distribution of a result (the original IEEE-754 result being drawn from this distribution). Although this method gives good results at scale [28], it is slow and can be complex to interpret. Indeed, one needs to run the computation several times, calling a random number generator for each operation, until a good estimator of the distribution of the output is reached. Overall results from the simulation are samples from a distribution, so the user still has to perform statistical analysis to conclude on the numerical quality of the result (CADNA [26] is a notable exception, encapsulating three synchronous runs to avoid re-runs and post-processing, but it uses a majority vote to evaluate comparisons that can impact the control flow of the program and the number of runs is fixed).

With our work, we offer a method with distinct properties: both fast enough to be used on large simulations and providing easily understandable results, which can be directly used and interpreted by end users. To do so, we observe that part of the work that has been done to increase the precision of computation, most notably pair arithmetic [30] (which derives from double-double arithmetic [1, 12]), can be repurposed and extended to model numerical error on the fly. Furthermore, it deals correctly with the cancelation problem that makes high-precision arithmetic an unwise choice to measure numerical error. To the best of our knowledge, we are the first to adapt those concepts to the measure of numerical error.

3 PROBLEM STATEMENT

Throughout the article, we assume that we are using IEEE-754 [25] floating-point arithmetic with the round-to-nearest (default) rounding mode.

IEEE-754 floating-point arithmetic uses a finite set of numbers to represent reals. Hence, some numbers have no exact representations, and the result of most arithmetic operations between floating-point numbers cannot be represented exactly as a floating-point number.

IEEE-754 floating-point arithmetic rounds the results to a number that can be represented with floating-point numbers (the nearest number that can be accurately represented when using the default rounding mode). A direct consequence is that operations are not associative in floating-point arithmetic. Those round-offs lead to visible differences between computations done with real numbers and floating-point numbers, we call those differences *numerical errors*.

It is essential to discriminate between three different types of numerical error:

- The *numerical error* is the difference between the result computed in infinite precision and the actual floating-point result. This quantity is entirely determined by a sequence of operations on given inputs (and not stochastic, unlike uncertainty) and, more importantly, *signed*. It is the quantity that we measure and manipulate in the following.
- The *absolute numerical error* is the absolute value of the numerical error. It is unsigned, and thus its manipulation over several operations can only lead to an upper bound.

- The *relative numerical error* is the absolute value of the numerical error divided by the result computed in infinite precision. IEEE-754 floating-point arithmetic operators try to minimize this quantity (by contrast, fixed-point arithmetic will try to minimize the absolute error).

Our goal is to give an estimation of the numerical error of the output of a given computation.

4 ENCAPSULATED ERROR: A NEW METHOD TO MEASURE NUMERICAL ERROR

We propose a new method, which we call *encapsulated error*, to evaluate floating-point accuracy, representing each number using a couple $(number, error)$ that contains both the result of the original IEEE-754 floating-point computations ($number$) and a *signed* first-order approximation of its current numerical error ($error$), with respect to the machine precision, such that their sum ($number + error$) is a first-order approximation of the result as obtained in infinite precision.

4.1 Error-Free Transformations

The main building block of our method is the use of error-free transformations. When using the rounding-to-nearest rounding mode, it is known [4] that the numerical error of the addition and multiplication operators in IEEE-754 floating-point arithmetic can be expressed exactly as a floating-point number without needing any additional precision. Error-free transformations, as introduced by Knuth [29, Section 4.4.2], are operations that, despite being build on floating-point operators, *provably* produce the exact numerical error for any floating-point operands (for a useful reference, see the work of Muller et al. [37, Sections 4.3, 4.4, 5.1, and 5.2]).

We use two transformations: *TwoSum* [29, Section 4.4.2, theorem B], an error-free transformation able to return the error δ_+ of the addition of two floating-point numbers x and y in the absence of overflow, and a *Fused Multiply-Add* (*fma*) used to extract the error δ_* of their multiplication (see Algorithm 2) but also the exact residual of the division and square root [37, Section 5.2].

ALGORITHM 1: TwoSum(x, y)

$z \leftarrow x + y$
$x' \leftarrow z - y$
$y' \leftarrow z - x'$
$\delta_+ \leftarrow (x - x') + (y - y')$
return δ_+

ALGORITHM 2: TwoMultFma(x, y)

$z \leftarrow x * y$
$\delta_* \leftarrow \mathrm{fma}(x, y, -z)$
return δ_*

It is to be noted that although we focus on the rounding-to-nearest rounding mode, there is a variation of the *TwoSum* error-free transformation that is valid for any rounding mode [40, Section 2.3] and that some proofs [20] show that the *Fused Multiply-Add* operation can still be used as an error-free transform for other rounding modes. Using those operations, one could generalize Algorithms 3 through 6 to other rounding modes.

4.2 Operations and Functions

Our operations take $(number, error)$ pairs, such as (x, δ_x) and (y, δ_y), as inputs and output a (z, δ_z) pair, where z is the result of the operation and δ_z its numerical error. δ_z is computed using both the error introduced by the operation and the error transmitted from the inputs (δ_x and δ_y).

During the evaluation of the arithmetic operations and the square root function, we compute the error produced by the operation using an error-free transformation. Then we combine it with the error transmitted from the inputs using basic arithmetic (Algorithms 3–6).

ALGORITHM 3: Addition((x, δ_x), (y, δ_y))

$z \leftarrow x + y$
$\delta_z \leftarrow \delta_x + \delta_y + \text{TwoSum}(x, y)$
return (z, δ_z)

ALGORITHM 4: Multiplication((x, δ_x), (y, δ_y))

$z \leftarrow x * y$
$\delta_z \leftarrow (\delta_x * y) + (\delta_y * x) + \text{fma}(x, y, -z)$
return (z, δ_z)

ALGORITHM 5: Division((x, δ_x), (y, δ_y))

$z \leftarrow x/y$
$numerator \leftarrow (\delta_x - \text{fma}(y, z, -x)) - z * \delta_y$
$denominator \leftarrow y + \delta_y$
$\delta_z \leftarrow numerator/denominator$
return (z, δ_z)

ALGORITHM 6: Square Root((x, δ_x))

$z \leftarrow \sqrt{x}$
$numerator \leftarrow \delta_x + \text{fma}(-z, z, x)$
$denominator \leftarrow z + z$
$\delta_z \leftarrow numerator/denominator$
return (z, δ_z)

Note that the second-order term, $\delta_x * \delta_y$, has been omitted from the computation of the error in the multiplication operation (which should be $(\delta_x * y) + (\delta_y * x) + (\delta_x * \delta_y) + \text{fma}(x, y, -z)$) and that we are linearizing the square root using its first-order Taylor approximation (see Section 5.2 for an analysis of the associated tradeoff).

The previous algorithms are equivalent to double-double arithmetic [1, 12] without the final re-normalization step and have been introduced independently by Latkin [31] and Lange and Rump [30]. Our work is different in that it gives a different semantic to this representation: our pairs model numbers and their errors rather than higher-precision numbers. This means that the error term should *never* impact the value of the numbers (hence the absence of the re-normalization step) or comparisons and the control flow of a computation.

For arbitrary functions, without any easy way to estimate the error introduced by the function, we deduce the final error by using higher-precision arithmetic (in practice, we use twice the working precision) to subtract the IEEE-754 arithmetic result from a higher-precision result computed after having corrected our number with its error (Algorithm 7).

The error bounds from Lange and Rump [30] apply to our arithmetic. They can be used to prove that, under their hypotheses and if we restrict ourselves to basic arithmetic operations (avoiding

ALGORITHM 7: Arbitrary Function$(f, (x, \delta_x))$

$z \leftarrow f(x)$
$\delta_z \leftarrow \{f(x + \delta_x) - z\}_{high\ precision\ computation}$
return (z, δ_z)

arbitrary functions that are not covered by Lange and Rump [30]), our computation of the numerical error is numerically stable. We test the accuracy of our evaluation on a practical case in Section 7.1.

4.3 Comparisons

To preserve the control flow of the original IEEE-754 computation, comparisons and tests are performed on the *number* part of the pair such that $(x, \delta_x) \leq (y, \delta_y) \Leftrightarrow x \leq y$.

Whenever we call a comparison operator, we can raise a warning if the subtraction of its arguments produces a number with no significant digits. Thus, it means that the comparison is numerically unstable (the implementation of CADNA [26] inspires this approach).

4.4 Outputs

Given the $(number, error)$ pair, our operations ensure that *number* is the result that would have been obtained with classical IEEE-754 floating-point arithmetic and *error* is an estimation of *number*'s numerical error.

We compute the number of significant digits of *number* in a base b using the following formula [39, Section 4.1]:

$$digits(number, error) = \begin{cases} \left\lfloor -\log_b \left| \frac{error}{number} \right| \right\rfloor & \text{if } number \neq 0 \text{ and } \left| \frac{error}{number} \right| \leq 1 \\ +\infty & \text{if } number = 0 \text{ and } error = 0 \\ 0 & \text{else} \end{cases}.$$

The formula has the interesting property of being quite resilient to imprecision in the measure of the error: as we round the logarithm of a ratio, having the correct order of magnitude for the error is enough to get the correct result.

5 ANALYSIS

5.1 Characteristics

Our method was built to have several key characteristics:

- It should give us an estimation of the *numerical error* of any number in a given computation so that a user can easily analyze an application, step by step if needed.
- Our outputs have to be pertinent for the non-instrumented application, meaning that the instrumented code should follow the same branches and code paths as the non-instrumented code and that the user should be able to confirm that the input was left unchanged by the instrumentation.
- We need to keep the runtime overhead low enough to ensure that our method is suitable for very large simulations.

Starting from those criteria, we designed the method around a simple principle: *computing the numerical error produced locally and combining it with the numerical error propagated from the inputs.*

Having those design criteria in mind, it is interesting to review some characteristics of the resulting method. As comparisons are computed on the *number* part of the pair, our method ensures that

we go through the same path and branches as the original computation. Therefore, it gives us both the same result that we would have had with the original computation in IEEE-754 arithmetic and a first-order approximation of its numerical error. Having both quantities for all numbers allows us to compute an estimate of the number of significant digits at all times and for all intermediate results.

The instrumentation might interfere with compiler optimizations, such as vectorization, causing the instrumented result to differ from the result of the original computation. However, this difference is bounded by the difference one observes between two levels of compiler optimization and can be quantified by checking whether the instrumented application's output is similar to the output of the non-instrumented application. In our experience, it does not interfere with the estimation of the numerical error.

Due to the fine granularity of the method, implementations can test the quality of the numbers produced after each operation and forward the information to a debugger to pinpoint the origin of the numerical inaccuracies in the computation (a functionality inspired by CADNA [26]).

Contrary to asynchronous methods (e.g., stochastic arithmetic), the control flow of the computation is not altered. Although this ensures that our analysis is pertinent to the un-instrumented application's outputs, we only explore and return information about the branches followed by the original computation. A more accurate result might have followed different branches (due to unstable comparisons) with vastly different behaviors: we know nothing of those behaviors. In practice, this has been observed to cause our method to underestimate the accuracy of some codes designed to be resilient to numerical errors in intermediate steps. Hence, it is crucial to be sure that meaningful tests are stable before starting to trust the estimation of the number of significant digits. Here, our method's ability to detect unstable branches is crucial.

Finally, our method is compatible with parallelism and, as we will present in the following sections, has a low overhead compared to state of the art. This makes it a suitable candidate to analyze high-performance computations.

5.2 Second-Order Terms

We could easily have added a second-order term to the formula for the multiplication operator (and deal with the square root as we do with arbitrary functions). However, we have observed that adding this term leads to either no sensitive improvement or even an artificial explosion of the numerical error. The most likely explanation is that the second-order term for the multiplication can be of the same magnitude as the numerical error introduced by the addition needed to incorporate it into our estimation.

If the computation of a number x is numerically stable, then, by definition, given the machine epsilon u ($u = \frac{1}{2}b^{1-m}$, the machine epsilon associated to an m digits floating-point system with base b), its numerical error is of order $o(|x| * u)$. Adding a second-order term to our estimate adds a correction of order $o(|x| * u^2)$, but the error introduced *by* the addition of the correction itself is of order $o((|x| * u + |x| * u^2) * u) = o(|x| * u^2 * (1 + u))$, which can dominate the correction, creating spurious results.

Meanwhile, if the numerical error is higher than $o(|x| * u)$, then the first-order approximation of the numerical error is already enough to reflect the fact that the computation is not numerically stable without requiring second-order terms.

Note that it should be possible to introduce this second-order term in a numerically stable way with an alternative formula based on a compensated dot product. Nevertheless, the impact on the computing time would be significant for, according to our tests, no sensitive improvement in accuracy.

5.3 Comparison with Higher-Precision Arithmetic

One might wonder how this scheme could be more precise than deducing the error from doing the computation with our target precision and higher-precision arithmetic side to side, especially since the usual rule of thumb for compensated summation and arithmetic relying on error-free transformations is to assume that they are equivalent to doubled precision.

Indeed, most operations get the same error estimation as with roughly doubled precision (minus any overlapping bits between the value of the number and its error). However, the decisive difference is in the resilience to cancelations. For instance, both 64-bit floating-point arithmetic and 200-bit floating-point arithmetic would evaluate $(2^{215} + 1) - 2^{215}$ as 0 (since even with 200 bits of precision, $2^{215} + 1$ rounds to the same binary representation as 2^{215}). A comparison between both results would conclude that 0 is the proper result and that there appears to be no numerical error. However, by keeping the numerical error as a separate quantity, the error introduced during the cancelation (1) is separated from the numbers that caused it, and thus the final error is still properly evaluated.

However, by keeping the numerical error as a separate quantity, encapsulated error keeps the error introduced during the cancelation (1) separated from the numbers that caused it, and thus the final error is still properly evaluated:

$$(\{10^{65}, 0\} + \{1, 0\}) - \{10^{65}, 0\} = \{10^{65}, 1\} - \{10^{65}, 0\} = \{0, 1\}.$$

This separation helps avoid the problems one encounters when selecting an insufficient precision to estimate the numerical error by comparison with an output computed in higher precision.

It is important to stress that our approach is optimized to keep track of the error and not increase the computation's precision. We could increase our implementation's precision by minimizing the number of overlapping bits between the two terms of the pair (the result would be similar to double-double arithmetic [12]). However, they would not represent the number obtained without instrumentation and its numerical error anymore. Thus, it would force us to run the computation a second time to evaluate the numerical error. By keeping a strict separation between the number and its numerical error, we make the computation of the numerical error efficient in terms of both numbers of operations and memory.

6 IMPLEMENTATION

One of the strengths of the method is its relative simplicity (the operations are independent, can be written in a few lines of codes, and build on basic arithmetic operators and a *Fused Multiply-Add* in the working precision), making it easy to produce a correct implementation in a programming language that supports operator overloading. We provide an optimized reference C++ implementation, the Shaman library [13], accompanied by tools to simplify the instrumentation and analysis of an application.

6.1 C++ Type Overload

The Shaman library relies on a custom numeric type with overloaded operations to instrument every arithmetic operation and function call. Our custom type uses the C++ template system to build an instrumented type on top of any IEEE-754 compatible type. Hence, we can define `Sdouble = S<double, double, long double>` to indicate that we want to use an instrumented type that behaves like the **`double`** type that is commonly used in C and C++ (numerical errors and implicit casts will be those of the **`double`** type), stores and manipulates numerical errors in a **`double`**, and does its higher-precision computations using the **`long double`** type (which is only used when computing an arbitrary operation function like a sinus).

Our implementation can also work with any IEEE-754 compatible type, as long as it rounds to nearest according to the IEEE-754 norm. This include the usual **`float`** and **`double`** types but also user-defined numerical types such as emulated 16-bit precision types. Furthermore, the user can use any type to manipulate the numerical error and do higher-precision computations.

Following our method to instrument arbitrary functions (Algorithm 7), our implementation includes all 73 mathematical functions in the current C++ standard library. We also provided an implementation of the C++ streaming operator, which prints numbers in scientific notation, displaying only significant digits. This is our preferred format to evaluate a result.

Our implementation is compatible with common parallelism framework such as OpenMP [7] (for which we provide reduce operations) and MPI [21] (for which we provide type definitions), but also with the Eigen [22] and Trilinos [23] linear algebra libraries (for which we provide appropriate type traits). These library-specific operations, definitions, and traits could be user defined, as their implementation within even a complex library such as Trilinos does not require access to a lower abstraction level.

6.2 Numerical Debugging

To help with Shaman's usage, we included a numerical debugger, inspired by CADNA [26] and made possible by our fine-grained error representation. The user can use GDB or any classical debugger to pinpoint specific types of unstable operations, such as cancelations and unstable comparisons.

The user can set a variety of compilation flags to indicate the kind of numerical errors of interest to him (e.g., unstable comparisons and cancelations). When one of those operations happens, the program calls the *instability* function. The user can set a breakpoint on the function, and it will pause the program giving access to all variables and their numerical error.

Although this lets the user quickly evaluate the localization of specific numerical errors, it might become overwhelming on large programs with thousands of cancelations and unstable tests, often due to only a small subset of operations. We resolved this problem by writing what we call a *numerical profiler*. It is a script that hooks itself onto GDB; records all breakpoint triggers; and produces a report with the line number, the operation name, and the number of occurrences. This automation script gives the user an overall view of the numerical behaviors of their application.

6.3 Clang-Based Automatic Refactoring

A library-based implementation, such as ours, lets users examine the instrumented code and compose our functions with their own to serve their specific purposes. However, this flexibility requires access to the source code and the manual replacements of all floating-point types in an application.

The opposite approach, binary instrumentation tools, such as Valgrind [38] and Intel PIN [33], can make the instrumentation of an application easier but obfuscate the instrumented code and are hard to compose.

We propose an automatic refactoring tool based on the Clang compiler [32], which offers a good compromise between both approaches. It takes a code, parses it with a state-of-the-art C++ compiler (Clang), includes Shaman's headers, matches and replaces types and functions in the abstract syntax tree, and outputs the instrumented code. The refactoring tool is careful to produce warnings where a human operator should review the code, such as in the interfaces of **`extern`** `C` sections.

7 EXPERIMENTATION

7.1 Accuracy

7.1.1 LU Factorization in Double Precision. To demonstrate our algorithm's accuracy, we applied it to the LU factorization of a double-precision 200×200 random matrix whose entries are

uniformly sampled from $[-1; 1]$. The LU factorization algorithm decomposes a matrix into a product of a lower triangular and an upper triangular matrix. It is a well-known algorithm that uses all arithmetic operators and is fundamental in linear algebra (used to solve linear systems, invert matrices, and compute determinants).

As the algorithm's numerical stability depends on the magnitude of its pivots, most implementations use the partial pivoting strategy. The test is performed twice, with and without partial pivoting. The condition number of our test matrix, defined as the ratio of its largest and smallest eigenvalues, is roughly 700. Thus, one would expect the numerical error to be sensibly smaller when using partial pivoting.

To evaluate the accuracy of Shaman's estimation of the error, we also run the LU factorization using MPFR and 10,000 bits of precision. Although using 10,000 bits of precision is too slow and memory intensive to be used on larger problems, it should be accurate enough to be used as a reference. Note that we set aside any cell with an infinite number of significant digits according to MPFR (meaning that MPFR and double precision reached the same number) when computing the averages.

Without any pivoting strategy, we measured an average of 13.03 significant digits and a mean absolute difference between our estimation and the 10,000-bit arithmetic estimation of 0.004 significant digits. With a partial pivoting strategy, we measured an average of 14.80 significant digits, suggesting that the algorithm is more numerically stable, and a mean absolute difference between our estimation and the 10,000-bit arithmetic estimation of 0.028 significant digits. In both cases, the difference between our estimation and the 10,000-bit arithmetic estimation of the numerical error is close to the machine epsilon (roughly 10^{-16} for 64-bit double precision).

Looking at Figure 1, one can observe that Shaman's estimation of the number of significant digits is accurate. It tends to be noisier when the number of significant digits is large, which makes sense, as it means that the numerical error is close to machine epsilon. However, its relative precision increases sensibly with the magnitude of the numerical error, when the number of significant digits decreases, making it suitable for the diagnosis of the numerical behavior of an application.

7.1.2 Integration by the Rectangle Rule in Float Precision. We also evaluated the accuracy of Shaman on the integration of the cosine function between 0 and $\frac{\pi}{2}$ using the rectangle method (this test case comes from the authors of VERROU [16] and, in particular, the work of Févotte and Lathuilière [17]).

In infinite precision, the only source of error is the discretization error of the integration, which reduces in $O(\frac{1}{n})$ as the number of rectangle, n, increases (meaning that the step size gets finer) until the result reaches 1, the known analytical value of the integral. In finite precision, here **`float`** precision (32 bits), there are two sources of error: the discretization error and the numerical error. Having an analytical solution is particularly interesting because it lets us evaluate our numerical error estimate directly, using the analytical solution instead of a reference method.

When we plot the difference between the result of the integration and the analytical value of the integral as a function of the number of rectangles (Figure 2), we observe that although it first decreases (which is predicted by the decrease in discretization error), it then starts to become noisier and increase.

This behavior is explained when we look at Shaman's estimation of the numerical error (here we display the absolute value of the raw numerical error and not the number of significant digits). It can be seen that the numerical error increases with the number of rectangles and ends up becoming the dominant source of error, the estimation of the numerical error perfectly overlapping with the error computed analytically.

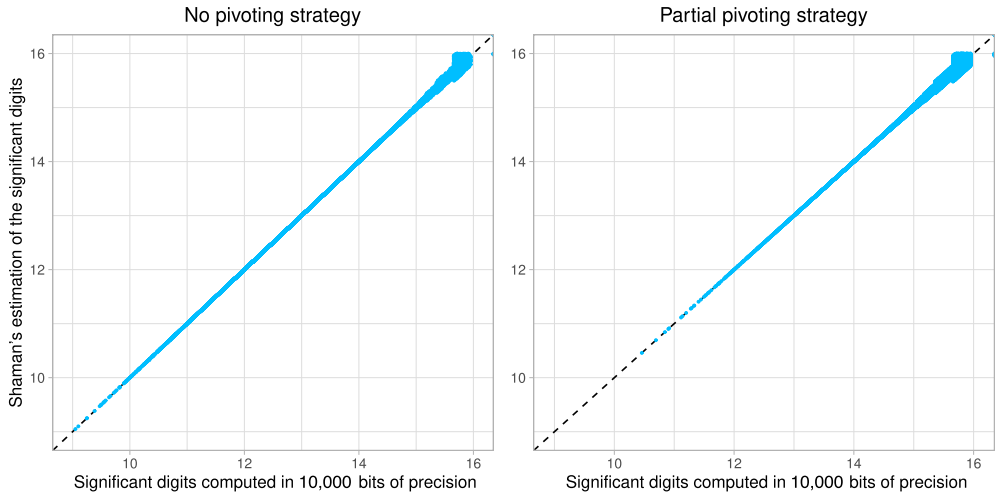

Fig. 1. Estimation of the number of significant digits for the LU factorization algorithm. Each dot corresponds to a cell in the non-zero half of either the L or U matrix produced by the decomposition. The dashed line represents the equality between both estimations of the number of significant digits.

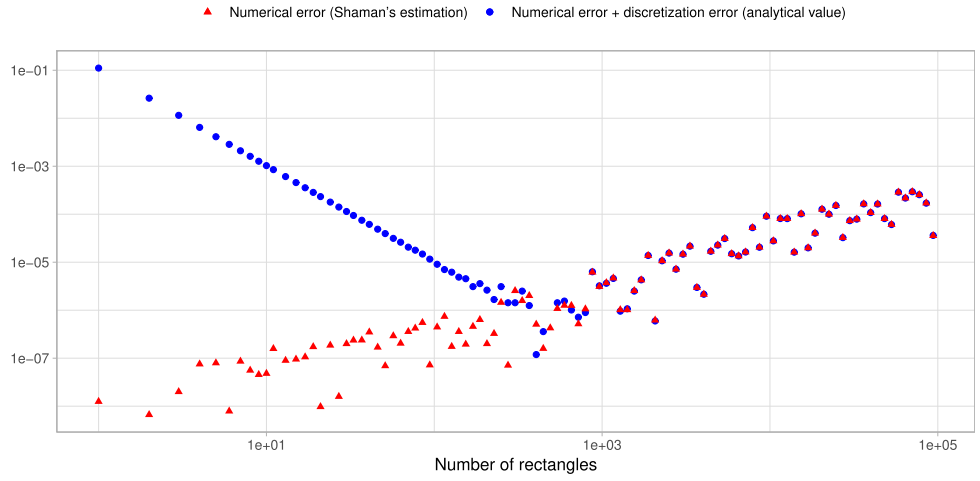

Fig. 2. Absolute value of the error as a function of the number of rectangles used for the integration of the cosine function between 0 and $\frac{\pi}{2}$ using the rectangle method. Both axes are displayed on a logarithmic scale.

7.2 Overhead

In this section, we evaluate the overhead of the Shaman library compared to state-of-the-art alternatives. We use highly optimized benchmark codes as well as a more realistic numerical application.

We picked one implementation for each of the most common approaches used to measure the numerical error. We choose these representatives because of their extensive usage in their category and efficiency:

- MPFR [15], with the MPFR C++ [24] wrapper, which implements arbitrary precision arithmetic (tested with 100 and 200 bits of precision).

- Boost Interval [5], which implements interval arithmetic.
- VERROU [16], which implements a form of stochastic arithmetic.
- CADNA [26], which implements a synchronous variant of stochastic arithmetic.

In the following, please note that for VERROU, we report computing time for only one run. However, stochastic arithmetic requires several runs to draw any conclusion. One would need to multiply VERROU's computing time by a factor of 5 or more to take this fact into account. Along the same line, we do not take into account the fact that to draw conclusions on the numerical error when using MPFR or any higher-precision arithmetic implementation, one would need to compare a higher-precision run with a double-precision run.

We instrumented four applications. The first three are the tasks that deal with floating-point arithmetic in the computer benchmark game [19], a well-known benchmarking suite that is used to compare the peak performance of programming languages on different tasks. The last application of our selection is the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics code (Lulesh 1.0 [27]), a proxy application for the performance benchmark for exascale computing. To sum up, our experiments consist of instrumenting and analyzing:

- *n-body*: *N*-body simulation.
- *Spectral norm*: Computing an eigenvalue using the power method.
- *Mandelbrot set*: Generating the Mandelbrot set at a given resolution, it is the only parallel benchmark of the set.
- *Lulesh 1.0*: Solving explicit hydrodynamics equations on a collection of volumetric elements.

All four applications are in double precision (represented by Shaman's `Sdouble` type). As the computer benchmark game provides several implementations for each task, we instrumented the fastest C++ implementations, at the time of writing, that do not rely on explicit vectorization or calls to libraries (such as Eigen) to do their computations. As the code is highly optimized, sometimes several orders of magnitude faster than a naive implementation, we expect to observe extreme behaviors on those applications.

We also instrumented the serial version of Lulesh 1.0 with a domain (mesh) size of 10^3 to compare different tools on an application that is representative of the kind of computations that are actually done in high-performance computing. We expect to observe a very representative overhead on this benchmark.

Each program was compiled with GCC 7.3.0 and the O3 optimization flag and ran on a four-core Intel Xeon CPU E3-1220 v3 @ 3.10 GHz with 16 GB of RAM. Measures presented in the following correspond to the minimum computing time over 30 runs (the average running time produces a similar plot, but it is less appropriate to estimate the intrinsic running time independent of the perturbations, as perturbations can only increase the running time). Since the variation coefficient was always below 2%, we do not include error bars for the sake of readability.

As shown in Figure 3, Shaman displays the lowest overhead on every application. Moreover, it is interesting to observe that although it takes eight operations to compute an addition with our formula (Algorithm 3), Shaman's slowdown stays well below a factor of 8 for most applications.

One explanation is that the computations spend a non-negligible time on non-numerical operations, but it seems unlikely given that those are numerically intensive applications. An alternative and more likely explanation is that since the compiler has full access to the instrumented code (which would not be the case if we were instrumenting at the binary level), and since there are no tests inside our arithmetic operators, the compiler can, and indeed does, still vectorize and use instruction-level parallelism. Furthermore, since our representation takes only twice as much memory as the original floating-point representation, it increases the arithmetic intensity, which is beneficial on modern processors [47].

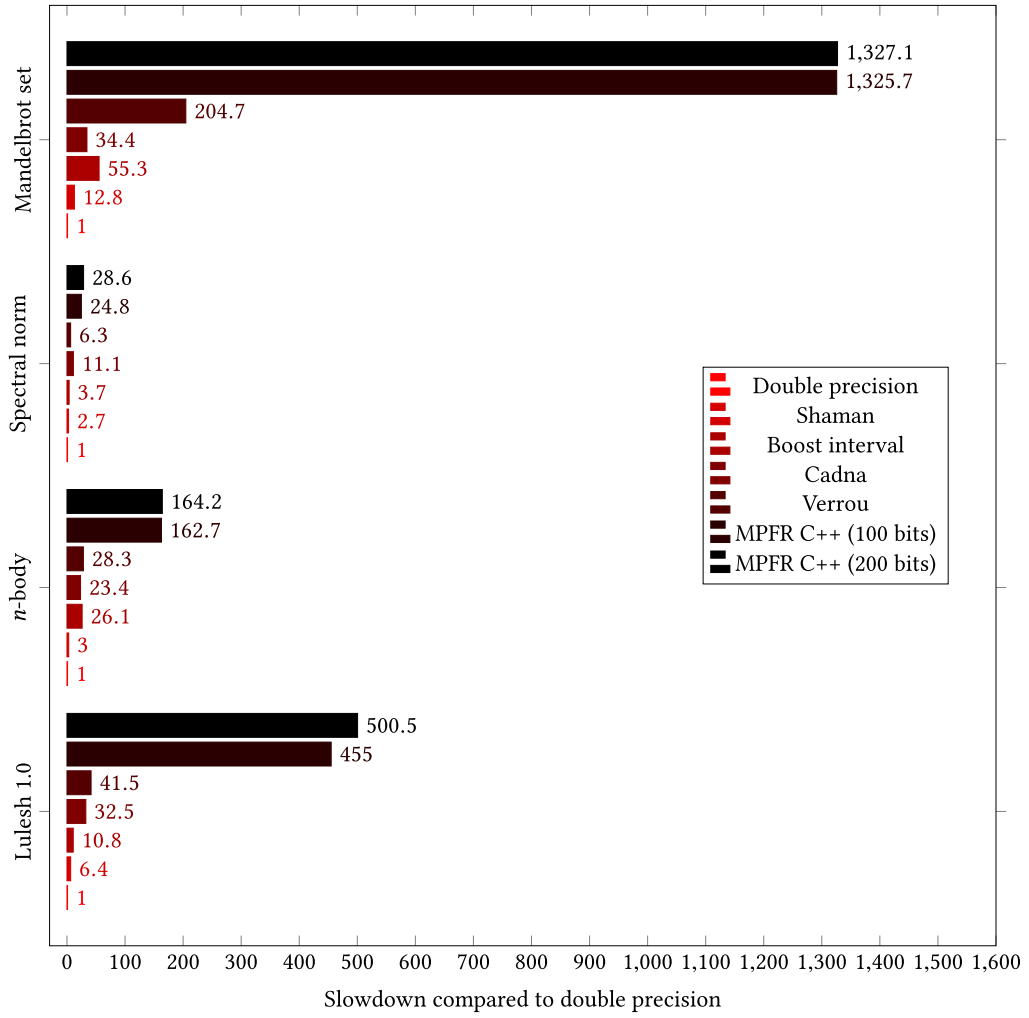

Fig. 3. Overhead of the Shaman library compared to the state of the art.

We note that the slowdown on the Mandelbrot set computation is significant for all tools, especially for MPFR C++ (we believe that MPFR C++'s overhead is dominated by the memory allocation and deallocation time, which explains why it varies very little from 100 bits to 200 bits). It might be due to the fact that it is the only parallel application of our benchmark or that its operations are highly optimized, making its performance more sensitive to source code modifications.

Finally, it is interesting to remark that the overhead observed on Lulesh 1.0 in this benchmark is consistent with the overhead we observed in most of the large numerical applications we have instrumented so far. Namely, Shaman's runs lead to a slowdown of 6 or 7 when profiling applications.

8 CONCLUSION AND PERSPECTIVES

In this article, we introduced the concept of encapsulated error, a new method to estimate an application's numerical accuracy and a reference implementation, the Shaman library.

Although it is a first-order estimation, we showed that our method is accurate and gives interpretable results. It can measure the number of significant digits of any result or intermediate result in a simulation and has a low overhead compared to state-of-the-art techniques. This combination of interpretability and low overhead is a conscious design decision to try to make floating-point analysis more accessible to authors of mathematical software. We are particularly interested in simulation and, more generally, high-performance floating-point computations, as they are sensitive to numerical errors but often produce applications unsuitable for analysis with pre-existing methods (as detailed in Section 2).

In the future, we would like to focus our attention on the localization of the sources of numerical error. Most publications focus on the localization of massive cancelations, but we believe that the progressive erosion in precision due to a large number of arithmetic operations can have an equally significant impact in the field of high-performance computing. FLUCTUAT [18] tries to approximate this quantity statically, whereas VERROU [16] proposes the use of delta-debugging to tackle this problem. However, inspired by affine arithmetic [6] and building on encapsulated error's fine granularity and direct access to the numerical error, we believe we could split the error term into one quantity per sub-section of the source code to collect precise information on the sources of numerical error in a single run of the application that is studied. This is something that cannot be done with existing approaches.

REFERENCES

[1] David H. Bailey. 1995. A Fortran 90-based multiprecision system. *ACM Transactions on Mathematical Software* 21, 4 (1995), 379–387.

[2] David H. Bailey and Jonathan M. Borwein. 2015. High-precision arithmetic in mathematical physics. *Mathematics* 3, 2 (2015), 337–367.

[3] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A dynamic program analysis to find floating-point accuracy problems. *ACM SIGPLAN Notices* 47, 6 (June 2012), 453–462.

[4] Gerd Bohlender, Wolfgang Walter, Peter Kornerup, and David W. Matula. 1991. Semantics for exact floating point operations. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*. IEEE, Los Alamitos, CA, 22–26.

[5] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. 2006. The design of the Boost interval arithmetic library. *Theoretical Computer Science* 351, 1 (2006), 111–118.

[6] João Luiz Dihl Comba and Jorge Stolfi. 1993. Affine arithmetic and its applications to computer graphics. In *Proceedings of the 7th Brazilian Symposium on Computer Graphics and Image Processing (Sibgrapi'93)*. 9–18.

[7] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55.

[8] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. Daisy-framework for analysis and optimization of numerical programs (tool paper). In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 270–287.

[9] Eva Darulova and Viktor Kuncak. 2014. *On Numerical Error Propagation with Sensitivity*. Technical Report. Ecole Polytechnique Federale de Lausanne. http://infoscience.epfl.ch/record/200132.

[10] Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. *ACM SIGPLAN Notices* 49, 1 (2014), 235–248.

[11] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. 2011. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers* 60, 2 (2011), 242–253.

[12] Theodorus Jozef Dekker. 1971. A floating-point technique for extending the available precision. *Numerische Mathematik* 18, 3 (1971), 224–242.

[13] Nestor Demeure, Cédric Chevalier, Christophe Denis, and Pierre Dossantos-Uzarralde. 2022. Shaman GitLab Repository. Retrieved February 24, 2023 from https://gitlab.com/numerical_shaman/shaman/-/tree/paper.

[14] Marco A. Feliú, Mariano Moscato, and César A. Muñoz. 2018. An abstract interpretation framework for the round-off error analysis of floating-point programs. In *Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science, Vol. 10747. Springer, 516–537.

[15] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software* 33, 2 (2007), 13–es.

[16] François Févotte and Bruno Lathuilière. 2016. VERROU: A CESTAC evaluation without recompilation. In *Proceedings of the International Symposium on Scientific Computing, Computer Arithmetics, and Verified Numerics (SCAN'16)*.
[17] François Févotte and Bruno Lathuilière. 2017. Verrou Tutorial for the PRECIS Summer School [in French]. Retrieved February 24, 2023 from https://github.com/edf-hpc/verrou/tree/ecole-precis.
[18] Eric Goubault. 2013. Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In *Static Analysis*. Lecture Notes in Computer Science, Vol. 7935. Springer, 1–3.
[19] Isaac Gouy. 2020. The Computer Language Benchmarks Game. Retrieved January 10, 2020 from https://benchmarksgame-team.pages.debian.net/benchmarksgame/.
[20] Stef Graillat, Jean-Luc Lamotte, and Diep Nguyen Hong. 2009. Error-free transformation in rounding mode toward zero. In *Numerical Validation in Current Hardware Architectures*. Springer, 217–229.
[21] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 22, 6 (1996), 789–828.
[22] Gaël Guennebaud and Benoît Jacob. 2010. Eigen v3. Retrieved February 24, 2023 from http://eigen.tuxfamily.org.
[23] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, et al. 2005. An overview of the Trilinos project. *ACM Transactions on Mathematical Software* 31, 3 (2005), 397–423.
[24] Pavel Holoborodko. 2010. MPFR C++. Retrieved February 24, 2023 from http://www.holoborodko.com/pavel/mpfr/. http://www.holoborodko.com/pavel/mpfr/.
[25] IEEE Microprocessor Standards Committee. 2019. *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2019 (Revision of IEEE 754-2008). IEEE, Los Alamitos, CA.
[26] F. Jézéquel and J.-M. Chesneaux. 2008. CADNA: A library for estimating round-off error propagation. *Computer Physics Communications* 178, 12 (2008), 933–955.
[27] Ian Karlin. 2012. *Lulesh Programming Model and Performance Ports Overview*. Technical Report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA.
[28] Gerald Knizia, Wenbin Li, Sven Simon, and Hans-Joachim Werner. 2011. Determining the numerical stability of quantum chemistry algorithms. *Journal of Chemical Theory and Computation* 7, 8 (2011), 2387–2398.
[29] Donald Knuth. 1998. *The Art of Computer Programming: Seminumerical Algorithms*. Vol. 2. Addison Wesley, Reading, MA.
[30] Marko Lange and Siegfried M. Rump. 2020. Faithfully rounded floating-point computations. *ACM Transactions on Mathematical Software* 46, 3 (2020), 1–20.
[31] Evgeny Latkin. 2014. Twofold fast arithmetic. *arXiv preprint arXiv:1401.6235* (2014).
[32] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *Proceedings of the BSD Conference (BSDCan'08)*, Vol. 5.
[33] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices* 40, 6 (2005), 190–200.
[34] Victor Magron, George Constantinides, and Alastair Donaldson. 2017. Certified roundoff error bounds using semidefinite programming. *ACM Transactions on Mathematical Software* 43, 4 (Jan. 2017), Article 34, 31 pages. https://doi.org/10.1145/3015465
[35] Ramon E. Moore. 1963. *Interval Arithmetic and Automatic Error Analysis in Digital Computing*. Stanford University, Stanford, CA.
[36] Ramon E. Moore. 1966. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ.
[37] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. 2010. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, Boston, MA.
[38] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices* 42, 6 (2007), 89–100.
[39] Douglass Stott Parker. 1997. *Monte Carlo Arithmetic: Exploiting Randomness in Floating-Point Arithmetic*. Technical Report. Computer Science Department, University of California, Los Angeles.
[40] Douglas M. Priest. 1992. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. Ph.D. Dissertation. University of California, Berkeley.
[41] Nathalie Revol. 2003. Interval Newton iteration in multiple precision for the univariate case. *Numerical Algorithms* 34, 2-4 (2003), 417–426.
[42] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC'13)*. 1–12.

[43] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding root causes of floating point error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 256–269.

[44] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2015. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. In *FM 2015: Formal Methods*. Lecture Notes in Computer Science, Vol. 9109. Springer, 532–550.

[45] Richard Stallman, Roland Pesch, and Stan Shebs. 2002. *Debugging with GDB*. Free Software Foundation Inc.

[46] Jean Vignes and M. La Porte. 1974. Error analysis in computing. *Information Processing* 30 (1974), 377–390.

[47] C. Yang, R. Gayatri, T. Kurth, P. Basu, Z. Ronaghi, A. Adetokunbo, B. Friesen, et al. 2018. An empirical roofline methodology for quantitatively assessing performance portability. In *Proceedings of the 2018 IEEE/ACM International Workshop on Performance, Portability, and Productivity in HPC (P3HPC'18)*. 14–23.

Received 30 October 2019; revised 27 May 2022; accepted 13 July 2022