Floating-point profiling of ACTS using Verrou

Hadrien Grasland[1,*], *François* Févotte[2], *Bruno* Lathuilière[2], and *David* Chamont[1]

[1]LAL, Univ. Paris-Sud, CNRS/IN2P3, Université Paris-Saclay, Orsay, France
[2]EDF R&D - Dept PERICLES - Palaiseau, France

Abstract. Floating-point computations are ubiquitous in scientific computing. Achieving high numerical stability in these computations affects not just correctness, but also computing efficiency, by accelerating the convergence of iterative methods and expanding the available choices of precision.
The ACTS project aims at establishing an experiment-agnostic track reconstruction toolkit. It originates from the ATLAS Run2 tracking software and has already received strong adoption by FCC-hh. It is also being evaluated for possible use by the CLICdp and Belle 2 experiments.
In this study, Verrou, a Valgrind-based tool for dynamic instrumentation of floating-point computations, was applied to the ACTS codebase for the dual purpose of evaluating its numerical stability and investigating possible avenues for use of reduced-precision arithmetic.

Introduction

Floating-point computations in HEP

Even though it is the closest programmatic equivalent of manual approximate computation, floating-point arithmetic is widely regarded as an arcane and complex topic [1]. As a result, non-expert users tend to handle it through imperfect heuristic approaches, including assuming that floating-point numbers are equivalent to real numbers, or favoring use of double precision, the most precise form of IEEE-754 arithmetic that has broad hardware support.

Double-precision computation, however, is neither necessary nor sufficient as an approximation of exact computation. It can be insufficient in the presence of numerically unstable algorithms which expose its inexact nature, including through accumulation of many small numbers in an indefinitely growing accumulator, or subtraction of two numbers of similar magnitude. And it can be unnecessary in the presence of input data which is far less precise than its relative accuracy (around 10^{-16}) and sufficiently stable algorithms.

On modern computer hardware, and in the face of the computational challenges of future HEP experiments, there is a strong incentive to prefer use of single-precision IEEE-754 arithmetic, which has broader hardware support, uses twice less memory resources (bandwidth, caches...), reduces the need for many internal iterations in IEEE-754-compliant operations, and enables wider vectorization. But determining where this precision is applicable can be challenging, especially in large codebases which were not designed for it.

As we shall see, dynamic program instrumentation can help address these challenges.

*e-mail: grasland@lal.in2p3.fr

ACTS (A Common Tracking Software)

ACTS [2] is a free and open-source software project for track reconstruction in high-energy physics (HEP) experiments. As a modernized version of the particle tracking code used by the ATLAS experiment [3–5] during Run 2 of the Large Hadron Collider, the project is focused on adoption of modern C++ standards, usability in multi-threaded workflows, and extended use of vectorization. Key features include:

- Constructing a tracking geometry description from TGeo, DD4Hep, or GDML input
- A simple and efficient event data model
- Performant and highly flexible algorithms for track propagation and fitting
- Basic seed finding algorithms

A key aim goal of the project is to support the ever increasing needs of future accelerators such as HL-LHC [3, 4, 6] and FCC [7].

1 Verrou: A dynamic floating-point instrumentation tool

Verrou [8] is a tool aiming at helping diagnose, debug and optimize FP-related issues in large, industrial scientific computing codes. As an example, Verrou has already been used to debug `code_aster`, a structural mechanics simulation tool of more than 1.2M lines of code [9].

From a user perspective, Verrou performs Dynamic Binary Instrumentation (DBI) on the analyzed program, replacing each FP instruction with a variant implementing another type of arithmetic. Program execution is otherwise left unperturbed, meaning that results are output like in any standard execution. Analyzing the observed results changes allows estimating the global impact of FP arithmetic during program execution.

Verrou performs this instrumentation by building on top of Valgrind. This enables high usability, as there is no need to manually change the program or even recompile it; all that is needed is to prefix the usual command with an invocation of Valgrind with the Verrou tool:

```
valgrind --tool=verrou [VERROU_ARGS]   PROGRAM [ARGS]
```

It is worth noting that DBI in general (and Valgrind in particular) naturally composes well with other tools which might be used in the analyzed program: compilers and specific compilation options, (potentially closed-source) third-party libraries, parallelization frameworks...

1.1 Floating-Point Arithmetic variants

In the command above, `[VERROU_ARGS]` allows changing several aspects in the behavior of Verrou, and most notably the type of alternate FP arithmetic which is to be introduced in the analyzed program. Verrou implements a few variants of FP arithmetic:

Fixed rounding mode: in this mode, the rounding mode can be fixed to any of the 4 standard rounding modes defined by IEEE-754: rounding upwards, downwards, toward zero or to the nearest FP number. Additionally, a non-standard rounding mode can be emulated, which always rounds an FP calculation in the opposite way to the standard rounding to nearest. As discussed in [10], some insight on FP errors can be obtained from the comparison of a few results obtained with the same program, using different rounding modes.

Stochastic rounding mode: in this mode, the result of every FP instruction is randomly rounded upwards or downwards. Depending on the chosen probability law, this can be similar to an asynchronous CESTAC arithmetic [11] or a variant of Monte-Carlo arithmetic [12]. Numerous works in the literature detail the statistical post-processing techniques which can

be used to assess the impact of FP arithmetic by analyzing the results of several randomly rounded executions of the program [12–14].

Single precision: in this mode, the result of every double-precision operation is rounded as if it had had been performed in single precision. This feature can be an easy way to simulate a single-precision version of any program, without having to change its source code.

1.2 Instrumentation scope

Algorithms which are not robust to changes in the FP rounding modes often indicate issues in the analyzed program: for example, such algorithms will produce varying results if a code parallelization and/or vectorization changes its execution order. However, some algorithms rightfully demand that a standard rounding be used for their correct execution. This is for example the case for trignonometric functions of the standard mathematical library (`libm`), which require rounding to the nearest.

In order to handle such cases, Verrou features the ability to restrict the scope of the instrumentation: it can be instructed to leave some parts of the program unperturbed. This can be performed at the granularity of functions (or, rather symbols in the object files and libraries composing the instrumented executable binary), but also at the granularity of source code lines (if debugging information are available[1]).

Like Valgrind, Verrou only instruments the program under study by default. But in multi-process applications, like some test runners, instrumenting child processes can be required. This can be done by adding the `-trace-children=yes` flag to the command above.

1.3 Debugging

While simply changing the underlying arithmetic can help assessing and diagnosing FP-related issues in a given program, more is needed in order to help debugging them. The aforementioned ability to restrict the scope of Verrou's instrumentation can be used to implement a form of bisection in order to find out which parts of the program are the most unstable. In order to automate such a search, the `verrou_dd` utility implements in the Verrou ecosystem several variants of the Delta-Debugging algorithm [15, 16]. These algorithms differ in subtle ways in the definition of "unstable" program parts, but it is enough in the following discussion to assume that "unstable" program parts are parts of which the perturbation is correlated to large changes in program results. The full details are given in [17].

1.4 Setting up a Verrou environment

Verrou is an open-source tool, which can be freely downloaded from:

`http://github.com/edf-hpc/verrou`

The simplest way of installing Verrou for general use is to download and build a combined stable release of Valgrind and Verrou. These are available from the "Release" tab of the GitHub page of Verrou.

As discussed in section 1.3, it is very useful, though not mandatory, to have debugging informations available for the application under study. This can be done by installing debug information packages for all numerical libraries used by the program, and by compiling the program itself with debugging information. In the case of CMake-based packages like ACTS, the latter can by done by using a `CMAKE_BUILD_TYPE=Debug` or

[1] As produced for example by using the `-g` command-line switch with `gcc`.

`CMAKE_BUILD_TYPE=RelWithDebInfo` build configuration. The former enables most precise numerical instability reporting, but at a significant execution performance cost, while the latter is a compromise between execution speed and application serviceability.

As a last setup step, it is useful to prepare a standard Verrou exclusion file. As explained in section 1.2, this exclusion list should arrange for Verrou to leave at least the standard mathematical library (`libm`) unperturbed.

2 Analyzing the numerical stability of ACTS

2.1 Instrumenting the ACTS test suite

As a first step, Verrou's random rounding mode was used to evaluate the numerical stability of the ACTS codebase. This was done by running the ACTS unit test suite and integration tests in Verrou, using random rounding mode, and checking which tests fail and why.

Most of the observed test failures emerged from dubious floating-point data handling practices in the test suite itself. For example, some unit tests would exactly compare the output of a numerical computation to an expected result (as on fig. 1), and could thus only succeed if ACTS' implementation exactly matched the test code. These exact comparisons were advantageously replaced with approximate comparisons, which provide more implementation freedom and can encode precision expectations more accurately, except in areas such as serialization where exact reproducibility would truly be expected. Similar issues included relative comparison of floating-point data with zero (which only succeeds if the input data is exactly zero) and unrealistic amounts of significant digits in textual data dumps.

```
b475987fe27f:~/acts-core/spack-build # valgrind --tool=verrou --rounding-mode=random --demangle=no                    \
>                                    --exclude=`pwd`/excludes.ex --trace-children=yes    ctest -V
                                                [...]
66: Running 4 test cases...
66: /root/acts-core/Tests/Utilities/InterpolationTests.cpp(37): error: in "interpolation_1d": check interpolate(Point({{2.3}}
), low, high, v) == 23. has failed [22.999999999999996 != 23]
66: /root/acts-core/Tests/Utilities/InterpolationTests.cpp(55): error: in "interpolation_2d": check interpolate(Point({{1.8,
1.}}), low, high, v) == 18. has failed [18.000000000000004 != 18]
66: /root/acts-core/Tests/Utilities/InterpolationTests.cpp(56): error: in "interpolation_2d": check interpolate(Point({{1.3,
3.}}), low, high, v) == 33. has failed [33.000000000000007 != 33]
66: /root/acts-core/Tests/Utilities/InterpolationTests.cpp(59): error: in "interpolation_2d": check interpolate(Point({{1., 1
.7}}), low, high, v) == 17. has failed [16.999999999999996 != 17]
66: /root/acts-core/Tests/Utilities/InterpolationTests.cpp(66): error: in "interpolation_2d": check interpolate(Point({{1.3,
1.7}}), low, high, v) == 20. has failed [19.999999999999996 != 20]
66: /root/acts-core/Tests/Utilities/InterpolationTests.cpp(68): error: in "interpolation_2d": check interpolate(Point({{1.8,
1.7}}), low, high, v) == 25. has failed [24.999999999999996 != 25]
66: /root/acts-core/Tests/Utilities/InterpolationTests.cpp(69): error: in "interpolation_2d": check interpolate(Point({{1.8,
2.5}}), low, high, v) == 33. has failed [33.000000000000007 != 33]
66: /root/acts-core/Tests/Utilities/InterpolationTests.cpp(102): error: in "interpolation_3d": check interpolate(Point({{1.3,
 2.1, 1.6}}), low, high, v) == 32. has failed [32.000000000000007 != 32]
66: /root/acts-core/Tests/Utilities/InterpolationTests.cpp(123): error: in "interpolation_mixed_point_values": check interpol
ate((p << 2.3).finished(), low, high, v) == 23. has failed [22.999999999999996 != 23]
66:
66: *** 9 failures are detected in the test module "interpolation tests"
```

Figure 1. ACTS unit test failure originating from exact comparison of floating-point results

2.2 Findings in the core ACTS library

Some test suite issues could inform future developments in ACTS' core library. For example, it was found that chaining affine transforms (as done when modeling these transforms as a combination of a translation and Euler rotations) could be very unstable on a numerical level. It could therefore be better to precisely compute the full transform matrix and pass it directly as a program input. This is quite intuitive: in the presence of large translations, a small error on a rotation angle will lead to a large difference in the output of an affine transform.

Other issues were also directly found in the core ACTS library. For example, one part of the codebase used to compute integer powers of two using the floating-point `pow` function, in

spite of this operation being computable more quickly and accurately using a bit shift. A few parts of the codebase would perform divisions whose denominators could get arbitrarily close to zero, without handling the numerical instability that ensues. And another function would compute the azimutal angle difference between two 3D vectors in Cartesian coordinates by converting both vectors to spherical coordinates, subtracting the spherical coordinates, and wrapping the result in the $[-\pi;\pi[$ range, when that could be done in a more efficient and precise way by leveraging vector product identities.

2.3 Edge cases and limits of this approach

One might expect that use of random rounding would lead to non-reproducible test failures. This occasionally proved true, especially in cases where the rounding of a single operation (rather than an accumulation of results) played an important role. However, Verrou comes well-prepared to handle such situations. It provides various alternatives to fully randomized rounding, including the deterministic rounding modes discussed in 1.1 and an option to control the random number generator seed. These features eliminate reproducibility issues in direct usage of Verrou, and reduce their occurrence in delta-debugging.

Overall, the ease with which Verrou allowed numerical stability issues to be discovered, located, and addressed is appreciable. However, the tool did also prove to have some false positives and prerequisites.

On the false positive front, trigonometry proved to be a problematic area, not just due to the `libm` issues described above, but also because approximate trigonometry features many edge effects, where a tiny rounding perturbation makes a great difference in the output. This manifested as angles suddenly jumping from $-\pi$ to π, breaking tests of binned spatial data structures; or as sines and cosines going beyond their valid range of $[-1;1]$ by a tiny amount, leading `NaNs`[2] to be produced when this data was passed to reciprocal functions.

When it comes to prerequisites, it quickly became clear that although delta-debugging can promptly narrow down some classes of issues, a detailed test suite that can precisely report the cause of failures and the numerical values involved remains an invaluable asset when the instability occurs in a utility function that is called in many different contexts.

3 Evaluating the viability of reduced precision

3.1 Challenges of reduced precision studies

After validating the correctness of ACTS' floating-point computations using Verrou, the next step was to evaluate the usability of single-precision computations in an ACTS context.

As discussed in the introduction, there is a strong efficiency incentive for using reduced floating-point precision whenever it is applicable. Sadly, like many physics libraries, the ACTS codebase was written to use double-precision everywhere as a safe default, with no easy way to configure another floating-point type. Modifying ACTS to make its floating-point precision configurable, even without suppressing the assumption that the precision be the same everywhere, is therefore a deep change in the codebase affecting most function interfaces and thousands of lines of code. Such a patch is hard to maintain over extended periods of time in such a fast-moving codebase, which led a previous attempt at extending use of single-precision computations in the ACTS codebase to fail.

Fortunately, however, this is an area where dynamic program instrumentation can help. Using the newly introduced reduced precision computation mode of Verrou, it became possible to do a large fraction of the single-precision ACTS validation studies without changing

[2]Not-a-number, an error value produced by IEEE-754 arithmetic when performing meaningless computations

a single line of code in the core ACTS library. Unit tests failures could easily be studied, and "trivial" failures that emerged from unrealistic test expectations (such as expecting single electron-volt resolution on the energy of output particles from a tera-electron-volt collision) could easily be corrected at the test suite level.

3.2 NaN backtraces: A case study

Verrou's new ability to locate occurences of `NaN` proved particularly useful as it enabled fixing a subtle ACTS bug that was not detected in the original double-precision studies.

ACTS allows use of interpolated magnetic field maps, which by definition are based on weighted averages of tabulated magnetic field data. Some data on the edge of the magnetic field table used to be left uninitialized, which was not detected because for valid accesses it always had zero weight in the interpolation. However, the property $0 \times x = 0$ does not always hold when x is an IEEE-754 floating-point number, due to the fact that $0 \times \texttt{NaN} = \texttt{NaN}$ and $0 \times \pm\infty = \texttt{NaN}$.

Since the probability of random uninitialized data being equal to `NaN` or $\pm\infty$ is much higher for single-precision numbers than for double-precision numbers ($\frac{1}{128}$ instead of $\frac{1}{1024}$, owing to the reduced exponent range), this resulted in the appearance of `NaN` in the output of the single-precision magnetic field interpolation test. The failure could be easily detected and understood using Verrou's new `NaN` debugging feature (fig. 2).

```
b475987fe27f:~/acts-core/spack-build # valgrind --tool=verrou --rounding-mode=float --demangle=no --exclude=`pwd`/excludes.ex
 --trace-children=yes --num-callers=50 ctest -V -R "BFieldMapUtils"
                                        [...]
67: ==2319== NaN:
67: ==2319==    at 0x5847E7F: _ZN5Eigen8internal4pmulIDv2_dEET_RKS3_S5_ (emmintrin.h:271)
67: ==2319==    by 0x585B570: _ZNK5Eigen8internal17scalar_product_opIddE8packetOpIDv2_dEEKT_RS6_S7_ (BinaryFunctors.h:89)
67: ==2319==    by 0x59A31C9: _ZNK5Eigen8internal16binary_evaluatorINS_13CwiseBinaryOpINS0_17scalar_product_opIddEEKNS_14Cwis
eNullaryOpINS0_18scalar_constant_opIdEEKNS_6MatrixIdLi2ELi1ELi0ELi2ELi1EEEEEESA_EENS0_10IndexBasedESE_ddE6packetILi16EDv2_dEET
0_ll (CoreEvaluators.h:727)
                                        [...]
67: ==2319==    by 0x5BDE2D4: _ZN5Eigen6MatrixIdLi2ELi1ELi0ELi2ELi1EEaSINS_13CwiseBinaryOpINS_8internal13scalar_sum_opIddEEKN
S3_INS4_17scalar_product_opIddEEKNS_14CwiseNullaryOpINS4_18scalar_constant_opIdEEKS1_EESC_EESG_EEEERS1_RKNS_9DenseBaseIT_EE (
Matrix.h:225)
67: ==2319==    by 0x5BDAD82: _ZN4Acts6detail16interpolate_implIN5Eigen6MatrixIdLi2ELi1ELi0ELi2ELi1EEES4_St5arrayIdLm2EES6_Lm
1ELm4EE3runERKS4_RKS6_SB_RKS5_IS4_Lm4EE (interpolation_impl.hpp:133)
67: ==2319==    by 0x5BD59E8: _ZN4Acts11interpolateIN5Eigen6MatrixIdLi2ELi1ELi0ELi2ELi1EEELm4ES3_St5arrayIdLm2EES5_vEET_RKT1_
RKT2_RKT3_RKS4_IS6_XT0_EE (Interpolation.hpp:95)
                                        [...]
67: ==2319==    by 0x4F6D54: _ZNK4Acts21InterpolatedBFieldMap11FieldMapperILj2ELj2EE8getFieldERKN5Eigen6MatrixIdLi3ELi1ELi0EL
i3ELi1EEE (InterpolatedBFieldMap.hpp:166)
67: ==2319==    by 0x477030: _ZN4Acts4Test15bfield_creation11test_methodEv (BFieldMapUtilsTests.cpp:88)
67: ==2319==    by 0x474767: _ZN4Acts4TestL23bfield_creation_invokerEv (BFieldMapUtilsTests.cpp:25)
67: ==2319==    by 0x549B9B: _ZN5boost6detail8function22void_function_invoker0IPFvvEvE6invokeERNS1_15function_bufferE (functi
on_template.hpp:118)
                                        [...]
67: ==2319==    by 0x43F2FE: _ZN5boost9unit_test9framework3runEmb (framework.ipp:1629)
67: ==2319==    by 0x463F10: _ZN5boost9unit_test14unit_test_mainEPFPNS0_10test_suiteEiPPcEiS4_ (unit_test_main.ipp:247)
67: ==2319==    by 0x4648CB: main (unit_test_main.ipp:303)
                                        [...]
67: Running 12 test cases...
67: /root/acts-core/Tests/Utilities/BFieldMapUtilsTests.cpp(105): error: in "bfield_creation": difference{-nan} between value
2_rz.perp(){-nan} and bField2_rz.perp(){8} exceeds 1e-09%
```

Figure 2. NaN backtrace originating from the use of uninitialized data ACTS' interpolated magnetic field map. The unabriged stack trace is 42 frames long and goes through very complex layers of C++ abstraction: manually walking through the ACTS codebase to locate the point where a NaN is generated would have been a major reverse-engineering undertaking.

3.3 Looking ahead

Being able to quickly detect, understand, and resolve these issues without maintaining a large ACTS patch means that more time is available to focus on the more fundamental issues

raised by the use of single-precision floating-point numbers in HEP's demanding numerical environment.

For example, legit test failures are currently under investigation in the ACTS particle propagation mechanism, which integrates the equations of motion to follow particle track hypotheses throughout the particle detector in order to evaluate the quality of these hypotheses. This particle propagation mechanism operates under numerical constraints which lie at the edge of the abilities of single-precision computations (e.g. locating muons with a precision of a fraction of a milimeter after propagating them through the multi-meter radius of the ATLAS detector), and it is possible that some parts of the ACTS track parameter propagation code may need to continue operating in double precision or use a compensated algorithm.

Verrou's delta-debugging abilities will therefore be used to understand if ACTS' particle propagation code features specific "precision bottlenecks" which could be made to work in single precision through algorithmic improvements or focused use of extended precision. Should that turn out not to be the case, the ACTS track propagation codebase will need to keep using double precision internally. It is once again remarkable that one might be able to reach this conclusion and plan ahead a possible reduced-precision port of ACTS without needing to start by maintaining such a port over extended periods of time.

Conclusion

Summary

Future HEP experiments will face unprecedented data processing challenges. Opening up established particle tracking codebases to new users, extending their validation procedure to maximize their ability to face new constraints, and optimizing them to help them better utilize available computing resources, are all worthwhile activities in this perspective.

In this context, the Valgrind-based Verrou tool was used to stress-test the ACTS codebase with new forms of numerical validation, and is now being used to extend usage of reduced-precision arithmetic throughout this codebase. One goal of this study is that in the future, ACTS developers will be able to confidently use single-precision arithmetic as a safe default, validate the stability of the results through random rounding, and only fall back to less efficient double-precision arithmetic where it is truly necessary.

Future work

Most of the ACTS improvements that were discussed in part 2 of this study have been merged into the ACTS codebase. As discussed in part 3, the focus of this project is now on planning out a reduced-precision port of ACTS which will rationalize choices of floating-point precision throughout the codebase.

Beyond ACTS, this study revealed some future areas of improvement in the Verrou toolchain, and in particular its delta-debugging features. The functionality is currently less generally applicable than it could be, owing to the fact that it does not handle utility libraries well. It is planned to improve upon this situation in 2019 by adding call site path sensitivity to Verrou, allowing the tool to report in which caller context instrumenting a function is detrimental, and to do delta-debugging on stack traces rather than mere symbols.

Parallelization of the delta-debugging process would be another desirable feature, owing to the fact that use of Verrou serializes and slows down program execution significantly, while delta-debugging's need to test many instrumentation configuration is an obvious avenue for parallelization. A prototype of parallel delta-debugging is already available in development

versions of verrou, but a finalized parallelization scheme will need to await the ongoing migration of delta-debugging to the more powerful "recursive DDmin" algorithm.

In the shorter term, Verrou 2.1 will be released soon, providing a stable version of the experimental features that were used for this study, including detailed reporting of `NaN` values, and on-the-fly conversion of intermediary program results to reduced precision.

TODO: Acknowledgements

TODO: References (pb: les titres des articles n'apparaissent pas)

References

[1] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys **23** (1991)

[2] http://acts.web.cern.ch/

[3] A. Collaboration, *Technical Design Report for the ATLAS Inner Tracker Strip Detector*, Tech. Rep. CERN-LHCC-2017-005. ATLAS-TDR-025, CERN, Geneva (2017), `https://cds.cern.ch/record/2257755`

[4] A. Collaboration, *Technical Design Report for the ATLAS Inner Tracker Pixel Detector*, Tech. Rep. CERN-LHCC-2017-021. ATLAS-TDR-030, CERN, Geneva (2017), `https://cds.cern.ch/record/2285585`

[5] https://atlas.cern/

[6] http://hilumilhc.web.cern.ch/

[7] https://fcc.web.cern.ch/

[8] F. Févotte, B. Lathuilière, *Verrou: Assessing floating-point accuracy without recompiling*, working paper, `https://hal.archives-ouvertes.fr/hal-01383417`

[9] F. Févotte, B. Lathuilière, *Studying the Numerical Quality of an Industrial Computing Code: A Case Study on code_aster*, in *International Workshop on Numerical Software Verification (NSV)* (Heidelberg, Germany, 2017), pp. 61–80, ISBN 978-3-319-63501-9

[10] W. Kahan, *How futile are mindless assessments of roundoff in floating-point computation* (2006), web paper

[11] J. Vignes, *A stochastic arithmetic for reliable scientific computation*, Mathematics and Computers in Simulation **35**, 233 (1993)

[12] D. Stott Parker, *Monte Carlo arithmetic: exploiting randomness in floating-point arithmetic*, Tech. Rep. CSD-970002, University of California, Los Angeles (1997), `http://web.cs.ucla.edu/~stott/mca/CSD-970002.ps.gz`

[13] J.M. Chesneaux, J. Vignes, *On the robustness of the CESTAC method*, C. R. Acad.Sci. Paris **1**, 855 (1988)

[14] D. Sohier, P. De Oliveira Castro, F. Févotte, B. Lathuilière, E. Petit, O. Jamond, *Confidence intervals for stochastic arithmetic*, preprint, `https://hal.archives-ouvertes.fr/hal-01827319`

[15] A. Zeller, *Why Programs Fail* (Morgan Kaufmann, 2009), ISBN 978-0-12-374515-6

[16] A. Zeller, R. Hildebrandt, *Simplifying and isolating failure-inducing input*, IEEE Transactions on Software Engineering **28**, 183 (2002)

[17] F. Févotte, B. Lathuilière, *Debugging and optimization of HPC programs in mixed precision with the Verrou tool*, in *Computational Reproducibility at Exascale* (Dallas (TX), USA, 2018)