Studying the numerical quality of an industrial computing code: a case study on code_aster

François Févotte* and Bruno Lathuilière

EDF R&D, département PERICLES, 7 bd Gaspard Monge, 91120 Palaiseau, France,
{francois.fevotte, bruno.lathuiliere}@edf.fr

Abstract. We present in this paper a process which is suitable for the complete analysis of the numerical quality of a large industrial scientific computing code. Random rounding, using the Verrou diagnostics tool, is first used to evaluate the numerical stability, and locate the origin of errors in the source code. Once a small code part is identified as unstable, it can be isolated and studied using higher precision computations and interval arithmetic to compute guaranteed reference results. An alternative implementation of this unstable algorithm is then proposed and experimentally evaluated. Finally, error bounds are given for the proposed algorithm, and the effectiveness of the proposed corrections is assessed in the computing code.

Keywords: floating-point, numerical verification, random rounding

1 Introduction

EDF is France's main electric utility. Like several other industries, its internal processes rely heavily on numerical simulations, which are performed by Scientific Computing Codes (SCC). To name only an example: numerous SCCs are used to study the safety of nuclear power plants, or optimize their production. It is therefore important that both EDF itself, but also others – like nuclear safety authorities, have confidence in the results produced by these codes. To this end, all SCCs undergo a Verification & Validation (V&V) process, during which various sources of errors are evaluated:

- modeling errors, *i.e.* differences between "real life" and the mathematical objects used to represent it;
- mathematical approximations, *i.e.* differences due to the simplification of the mathematical problem (such as discretization for example) or to their approximate resolution (for example using iterative processes);
- computation errors, due to the difference between the ideal manipulation of real numbers, and actual computations performed by program running on a CPU, which typically uses floating-point arithmetic as standardized by the IEEE-754 norm [8].

* corresponding author

The first two sources of errors mentioned above have been studied for a long time, as they were the more dominant terms. However, continual progress in computational power over the last decades, as well as advances in numerical methods, have made it possible to dramatically increase the complexity of models, while at the same time improving their resolution (for example through refined discretizations). The impact of computing errors on results has therefore recently become of non-negligible importance, and the analysis of floating-point arithmetic is now a topic of interest for industry. However, as of today, the introduction of adequate methodologies in industrial V&V processes still largely remains to be done.

The main objective of the present paper is to describe a process which is suitable for the complete analysis of floating-point errors and numerical instabilities in a large industrial code such as code_aster.

Code_aster [1] is an open source scientific computation code dedicated to the simulation of structural mechanics and thermomechanics. It has been actively developed since 1989, mainly by the R&D division of EDF. It uses finite elements to solve models coming from the continuum mechanics theory, and can be used to perform simulations in a wide variety of physical fields such as mechanics, thermal physics, acoustics, seismology... Code_aster has a very large source code base, with more than 1 200 000 lines of code, mostly written in 3 languages: Fortran90 ($\simeq 60\,\%$), C ($\simeq 20\,\%$) and Python ($\simeq 20\,\%$). It also uses numerous third-party software, such as linear solvers or mesh manipulation tools. Its development team has been dedicated to code quality for a long time, and has accumulated several hundreds of test cases which are run frequently as part of the V&V process.

Despite this, the development team of code_aster has faced numerous non-reproducibilities and other errors thought to be related to floating-point arithmetic. The traditional methodology to identify and track such errors relied on the analysis of the robustness of the code to changes in its input parameters. Perturbing the mesh used in the discretization of the underlying Partial Differential Equations (PDEs) was a good way of performing such an analysis: results of the computation should be unaffected by the numbering of meshes, or almost unaffected by very small perturbations of the mesh nodes. Checking the robustness of computed results to such changes would allow to uncover errors related to numerical instabilities. The approach that we describe here is complementary: we study the robustness of the code with respect to the underlying arithmetic and perturbation of the computational process itself.

In the rest of this paper, we present a complete process allowing to study the numerical quality of code_aster, from evaluating its numerical stability (Sect. 2) and finding the origin of instabilities in the source code (Sect. 3), to fixing problems by proposing more stable algorithms (Sect. 4). The effectiveness of the proposed correction is then assessed in Sect. 5, before we make a few concluding notes in Sect. 6.

2 Checking for Numerical Instabilities with Verrou

Among the different techniques which can be used to evaluate numerical instabilities and round-off errors, the wide family of methods revolving around Monte-Carlo Arithmetic (MCA) [20] seems to be one of the most promising in industrial contexts. For example, Discrete Stochastic Arithmetic, as implemented in the CADNA library [9,11], has already been successfully used on large industrial codes [12]. However, the need for a complete instrumentation of the source code makes CADNA too costly a solution for it to be applied widely to industrial codes. Even less demanding tools such as Verificarlo [3], which only requires a re-compilation of all the source code (potentially including third-party libraries, if one wants to analyze them), are too impractical to be applied routinely as part of the V&V process of such a large code as code_aster.

In order to avoid the need for an instrumentation of the program sources or a recompilation, various tools aim at implementing numerical debugging in the form of a Dynamic Binary Analysis (DBA), *i.e.* by directly analyzing the executable binary program during its execution. For example, Craft HPC [10] is a tool performing DBA in order to detect cancellation errors, and evenmore optimize the use of single- and double-precision variables throughout the source code to balance speed and accuracy. Among DBA tools, many make the choice of leveraging the powerful DBA features of the Valgrind [13] platform. Such solutions are more advantageous since Valgrind is already used by a large number of scientific code developers to help with memory debugging, and is thus compatible with most computing codes.

FpDebug [2] is one of the earliest Valgrind-based floating-point analysis tools. It makes use of the "shadow memory" feature of Valgrind to perform a high-precision computation alongside the standard execution of the analyzed program. A comparison between high and standard precision results is performed to detect the occurrence of inaccuracies and, for each inaccuracy, to determine whether it comes from input data or from the floating-point operation itself. Apart from its very large overhead, mostly due to the use of shadow memory, the major problem preventing the use of FpDebug for industry-scale programs is the very large size of the output it produces. This problem should be tackled by Herbgrind [17], a promising tool which also uses shadow executions to detect floating-point inaccuracies, but uses advanced techniques to precisely track the origin of such errors in the source code. The Herbgrind output is therefore reduced to the set of code fragments which lead to inaccuracies, in a suitable form for later analysis with Herbie [16]. The large overhead induced by this in-depth analysis and the use of shadow memory in Valgrind — currently of the order of $\times 10$ to $\times 10\,000$ depending on the test case — makes the tool more suitable for the analysis of a single unstable test case, once the presence of numerical instabilities has already been uncovered (*cf.* Sect. 3).

2.1 Presentation of Verrou

The present work is based on Verrou [5,6], an open-source[1] floating-point arithmetic diagnostics tool developed by the R&D division of EDF. From the beginning of its development, Verrou has targeted large industrial applications by ensuring that basic diagnostics features can always work without recompiling the analyzed program nor having access to its source code.

Like other Valgrind-based tools, a major advantage of Verrou is its simplicity of use. When running a program, one only needs to add a prefix to the command line in order to instrument it:

```
valgrind --tool=verrou --rounding-mode=random PROGRAM [ARGS]
```

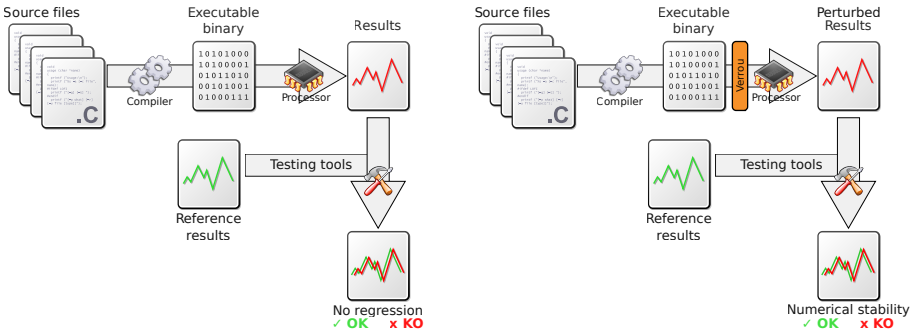
When called in this way, Verrou makes use of the Dynamic Binary Analysis (DBA) features of Valgrind to instrument the program (in binary form, as it was produced by its standard industrial build process) and to replace each floating-point instruction by a modified version which yields randomly rounded results. Global results of the computations are thus output like in any normal execution, except that they are affected by the cumulative effect of all randomly rounded intermediate results. As such, Verrou implements a Random Rounding Arithmetic (RRA), which might be seen either as a subset of MCA, or as a form of asynchronous CESTAC method [21].

This makes it easy for Verrou to be introduced in an industrial V&V process: as depicted in Fig. 1a, SCCs always have a non-regression test suite in which numerous test-cases are run to produce results which are compared to references. A tooling machinery often produces a nicely formatted synthesis of the results in order for the developers to see at a glance whether a change introduced in the code breaks something. Figure 1b presents how, by simply ensuring that test cases are run within Verrou, their results can be perturbed using RRA. Such results can then be compared as usual to references, in order to evaluate the numerical stability of the analyzed computing code.

The overhead of Verrou is kept as low as possible by a careful implementation of the various operations in directed rounding, limiting the number of random number generations, and using fused multiply-add (FMA) instructions when the hardware supports it. Overall, the slow-down factor for one execution of a given program in Verrou with respect to native run times is usually measured between $\times 10$ and $\times 20$. The most extreme overheads that we have measured so far were $\times 8$ (for a code spending much time in I/O operations) and $\times 40$ (observed in one test-case of an extremely well-optimized code). These factors then have to be multiplied by the number of random rounding runs needed for triggering anomalies; 3–5 is usually enough.

Verrou also provides more advanced features, such as the ability to limit instrumentation to parts of the code (functions, or even source lines if the binary was compiled using the "`-g`" switch).

[1] Project page URL: `http://github.com/edf-hpc/verrou`



(a) Standard non-regression testing (b) Numerical verification with Verrou

Fig. 1: Schematic view of industrial verification processes

2.2 Preliminary Work

In order to check that the instrumentation process itself does not introduce errors, it is interesting to perform a preliminary study. We work here on a subset of 72 test cases, covering a wide range of features in code_aster. We took care to include in this selection some test cases known (or believed) to be unstable, and some not known to be particularly unstable. For each of these test cases, we check that:

- several standard runs of code_aster yield reproducible results;
- several runs of code_aster within Valgrind (memcheck) yield results which are both reproducible and identical to the results of a standard run;
- several runs of code_aster within Verrou in nearest mode yield results which are both reproducible and identical to the results of a standard run.

This allows for an early detection of instrumentation problems, which could otherwise invalidate the conclusions of further studies. For instance, the origin of such problems include:

introspection: when part of the code examines its own execution (*e.g.* memory consumption or elapsed times) to make choices, it is to be expected that overheads induced by the instrumentation cause differences in results;

use of 80-bit instructions: the use of 80-bit instructions coming from the x87 set results in tricky non-reproducibilities, since intermediate results can be either kept in 80-bits registers, or stored as 64-bit double-precision numbers in memory. What actually takes place in the CPU can differ between native executions, and executions within Valgrind;

use of non-default floating-point arithmetic, *e.g.* directed rounding, or specific floating-point exceptions. These might not be correctly taken into account by Valgrind/Verrou.

For code_aster, these preliminary checks allowed to uncover a minor incompatibility between the instrumentation in Verrou and the `dgemv` routine from the OpenBLAS library. Investigations are under way to determine whether this behaviour is expected and understandable (*i.e.* this situation falls into one of the three categories mentioned above), or whether it is a bug. In the latter case, this could be either a bug in Verrou, such as an incorrect handling of some specific instruction used in this routine, or a real instability in OpenBLAS.

In the meantime, this routine has been replaced (using the `LD_PRELOAD` mechanism) by its equivalent from the Netlib implementation of the BLAS interface. Hence, the stability problem is temporarily curbed so as to focus on the analysis of code_aster itself.

2.3 Numerical Verification Using Random Rounding

We now perform the numerical verification of code_aster using random rounding with Verrou. For the sake of brevity, although this analysis has been performed on the 72 test cases mentioned above, Table 1 only presents the results for a few test cases. Each test case is identified by its name in the first column. The following 4 columns present the status of the run, as reported by the non-regression testing tools, respectively for a run under Verrou in nearest rounding mode, and 3 runs with Verrou in random rounding mode.

We present in the last column the number of significant (decimal) digits in common between the results of the three random rounding runs. This number is defined as

$$C(x) = \log_{10} \left| \frac{\mu(x)}{\sigma(x)} \right|,$$

where, for a sample $x = (x_1, x_2, \ldots, x_N)$, we denote by $\mu(x)$ its average, and $\sigma(x)$ its standard deviation. A star (*) denotes the fact that all digits output by code_aster were identical in the different runs in random rounding (*i.e.* in such cases, we have $\sigma(x) = 0$ in the above formula, but do not know whether this is due to results being "perfectly stable", or code_aster outputting too few digits). The column contains several numbers, as each test case performs non-regression testing on several results. For example, test case `ssls108i` outputs two results, each one of them being compared to a reference value. For the first one, the 3 random rounding runs produce values that have 11 decimal digits in common. Likewise, values produced by the 3 random runs for the second result have 10 decimal digits in common.

One (native) run of the test suite (72 test cases) takes around 10 minutes to complete, and each random-rounding run of the test suite with Verrou takes approximately 20 minutes. Therefore, the analysis presented here (one native and 3 random rounding runs) takes approximately 70 minutes to complete. The relatively low global overhead is explained by two factors:

- a large part of the test suite run time is spent in file-system manipulations and other I/O operations ;

Table 1: Analysis of numerical instabilities with Random Rounding

Test case	Status nearest	rnd$_1$	rnd$_2$	rnd$_3$	# common digits $C(\text{rnd}_1, \text{rnd}_2, \text{rnd}_3)$
`ssls108i`	OK	OK	OK	OK	11 10
`ssls108j`	OK	OK	OK	OK	10 10
`ssls108k`	OK	OK	OK	OK	11 10
`ssls108l`	OK	OK	OK	OK	10 9
`sdnl112a`	OK	KO	KO	KO	6 6 6 * 3 0
`ssnp130a`	OK	OK	OK	OK	* * 10 10 10 10 9 * * * 9 9 9 9 * * 10
`ssnp130b`	OK	OK	OK	OK	* * 11 11 * 12 9 * * * 9 9 9 9 9 * *
`ssnp130c`	OK	OK	OK	OK	* 11 11 11 11 10 9 11 11 10 10 10 * 11
`ssnp130d`	OK	OK	OK	OK	* 9 * * * 10 9 9 9 9 9 9 9 9 * 9 * * *

– the overhead of a random rounding run is relatively low, around $\times 10$ on average between test cases.

Over the 72 test cases used in this study, 3 exhibit an unstable behaviour as shown for example by `sdnl112a` in T5able 1: such tests may fail in random rounding mode and/or produce results that have very few significant digits in common between random rounding runs (3 or less).

In the rest of this paper, we focus on the further analysis of test case `sdnl112a`. Now that it has been shown to exhibit an unstable behaviour, the next logical step consists in trying to locate the origin of these instabilities within the source code, in order to correct them.

3 Locating the Origin of Numerical Errors in the Source Code

Verrou provides different ways to locate the origin of numerical errors, adapted to these different types of errors. A first technique, described in Sect. 3.1, allows the identification of functions and source code lines which produce large changes in the results when perturbed with random rounding. This is useful to locate the source of numerical errors but, in some instances, such sources of errors produce very small (and legitimate) inaccuracies, whose large impact on the final result only comes from an unstable test occurring later during program execution. In such instances, it is more appropriate to fix branching instabilities rather than the source of round-off errors. Another technique, described in Sect. 3.2, allows finding such unstable tests.

An interesting extension to the present study would consist in testing the relevance of high-overhead / high-fidelity tools such as Herbgrind, since the scope of the analysis at this stage is now reduced to one test case.

```
      do 60 jvec = 1, nbvect
          do 30 k = 1, neq
              vectmp(k)=vect(k,jvec)
30        continue
          if (prepos) call mrconl('DIVI', lmat, 0, 'R', vectmp,1)
          xsol(1,jvec)=xsol(1,jvec)+zr(jvalms-1+1)*vectmp(1)
          do 50 ilig = 2, neq
              kdeb=smdi(ilig-1)+1
              kfin=smdi(ilig)-1
              do 40 ki = kdeb, kfin
                  jcol=smhc(ki)
                  xsol(ilig,jvec)=xsol(ilig,jvec) + zr(jvalmi-1+ki) *&
                                        vectmp(jcol)
                  xsol(jcol,jvec)=xsol(jcol,jvec) + zr(jvalms-1+ki) *&
                                        vectmp(ilig)
40            continue
              xsol(ilig,jvec)=xsol(ilig,jvec) + zr(jvalms+kfin) *&
                                    vectmp(ilig)
50        continue
          if (prepos) call mrconl('DIVI', lmat, 0, 'R', xsol(1, jvec),&
                                        1)
60    continue
```

Fig. 2: Excerpt from the source code of function `mrmmvr`, which performs the product of a sparse matrix (whose non-zero coefficients are stored in array `zr`) with multiple vectors (stored in array `vect`). Vectors resulting from these products are stored in array `xsol`. Highlighted source code lines are those detected as unstable by the Delta-Debugging algorithm.

3.1 Delta-Debugging to locate round-off error sources

A first technique relies on Verrou's ability to restrict the scope of random rounding perturbations to parts of the program: functions, and possibly source code lines if the program was compiled with the right options (like `gcc -g` for instance). Starting from a situation where perturbing the whole program produces significant errors, this feature can be used to perform a binary search (based on the Delta-Debugging (DD) algorithm [22]) that progressively reduces the scope of instrumentation in order to eventually identify unstable portions of the source code, whose perturbation produces large changes in the results [5].

Performing an analysis of test case `sdnl112a` using the Delta-Debugging technique takes approximately 2 h 20 min. Overall, the Delta-Debugging algorithm tests 86 configurations (*i.e.* subsets of functions or lines which are perturbed). A configuration is considered correct if 15 random rounding runs pass the test suite criteria. Each random rounding run take approximately 10.3 s. to run (*vs.* 3.9 s. for a nearest rounding run).

In a first stage, the search identifies one unstable function, named `mrmmvr`. In a second stage, the Delta-Debugging search refines the localization and identifies 5 unstable lines, as shown in Fig. 2.

Although it might not be obvious at first sight, function `mrmmvr` performs the product between a sparse matrix M and several vectors v_k. Unstable lines

identified by the DD algorithm correspond to the dot products between each line $M[i,:]$ of the matrix and each vector $v_k[:]$. Such errors can be fixed relatively easily by introducing a compensated sum or dot product implementation. Numerous details can be found in the literature (see for example [14] or [15]), so we will not provide further details here.

3.2 Coverage analysis to locate unstable tests

An interesting side-effect of analyzing the binary is that Verrou is largely compatible with other forms of instrumentation based on source modification or recompilation. This composability helps devising a second localization methodology, which allows finding unstable tests. The technique consists in performing a coverage analysis of the test case in nearest rounding mode, and comparing it to the same coverage analysis performed during a random rounding run. Such a coverage analysis can be conducted using standard tools, such as using the `gcov` utility from the `gcc` suite. An example output of coverage analysis is presented in the left part of Fig. 3: the `gcov` tool produces annotated source files, where each line is prefixed by the number of times it was executed during the run. Dashes (`-`) indicate lines which do not contain executable code, and hashes (`#`) mark lines which were never executed.

The right part of Fig. 3 presents results of a coverage analysis performed in the same conditions, except that the program was perturbed with random rounding using Verrou. Lines highlighted in the figure are those for which the coverage count is different between the native run and the random rounding run. This identifies unstable tests, which led to different branches being taken.

This technique is much faster than the Delta-Debugging method presented above, since it only needs a few runs of the program: one standard run in nearest rounding mode, and as few random rounding runs as necessary to trigger instabilities (ideally only one, as it is the case here). A small additional overhead is due to the `gcov` instrumentation but, overall, the analysis of unstable tests in case `sdnl112a` takes less than a minute to complete.

Three unstable tests were found this way in code_aster, including the function illustrated in Fig. 3, on which we will focus in the rest of this paper. A quick inspection shows that the incriminated function aims at computing

$$f(a,b) = \begin{cases} a & \text{if } a = b, \\ \frac{b-a}{\log(b)-\log(a)} & \text{otherwise,} \end{cases} \tag{1}$$

which is a continuous function in real arithmetic, but whose current implementation in Fortran exhibits (not too unsurprisingly) unstable behavior in floating-point arithmetic. Our next step should then be to transform this formula into another expression, more stable when evaluated in floating-point arithmetic. Since the implementation presented in Fig. 3 uses the IEEE-754 *binary64* [8] format for all relevant variables (declared using `real(kind=8)` in Fortran), we will focus on this precision. The rest of this paper will therefore consistently use double-precision floating-point arithmetic, and define the relative rounding error as $\mathbf{u} = 2^{-53}$.

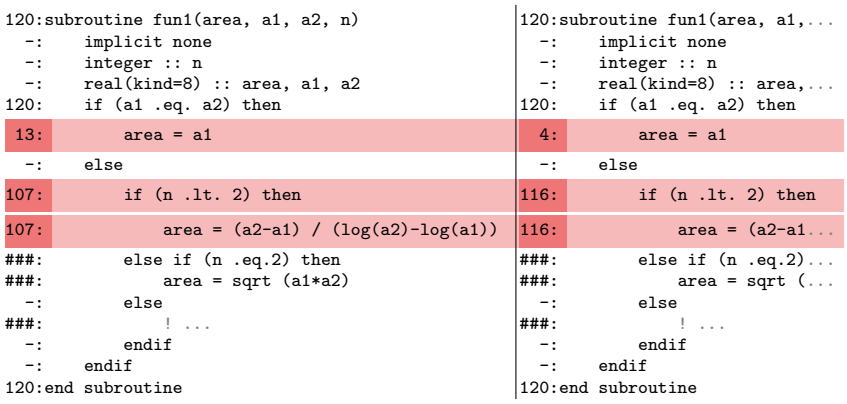


Fig. 3: Instability detection using code coverage diagnostic tools: results of a standard coverage diagnostic (left), compared to a coverage diagnostic perturbed with random rounding using Verrou (right). Highlighted source code lines are those for which the count of occurrences is different in both executions.

4 Fixing Floating-Point Instabilities

Instabilities such as the one uncovered here can quickly be studied by developing a small stand-alone application performing only the calculation of $f(a, b)$. Since the scope of the analysis is now reduced to a single code fragment, the field of applicable tools considerably widens. For example, one approach could consist in performing a static analysis of the code. However, all we interested in here is fixing the expression, which exactly what the Herbie tool [16] was designed to do. Herbie aims at providing more accurate replacements for expressions which are inaccurately evaluated in floating-point arithmetic. It does so by evaluating the input expression on a sample of all parameters in order to assess its accuracy by comparison between standard precision floating-point arithmetic and higher-precision arithmetic. Multiple transformations are then tested on the input expression in order to generate replacements, which are in turn evaluated to check whether they are more accurate than the initial one. Herbie tries by default to generate an expression that maximizes the average floating-point accuracy over the whole set of samples.

Unfortunately, in our case, expression (1) is very accurate almost everywhere in the space of parameters. The only inaccuracies occur when a and b are very close, which leads to a catastrophic cancellation between the logarithms in the denominator. Even increasing the number of points sampled by Herbie (100 000 points instead of the default 256), the subset of values for which a is close enough to b to cause error is not sampled. Therefore, even if we tell Herbie to optimize the worst case accuracy, the error is underestimated and no useful replacement is proposed. In [16], Herbie authors mention the need for a sampling of millions of points in order to correctly evaluate the worst-case accuracy of expressions with more than one argument.

Fig. 4: Relative errors on $f(a, b)$, as computed in floating-point arithmetic using the initial implementation found in code_aster, and the corrected implementation proposed in this work

Algorithm 1: Floating-point implementation of the proposed formula

Data: a, b
Result: $p \simeq f(a, b)$

$c \leftarrow b \oslash a$;
$n \leftarrow c \ominus 1$;
if $|n| \leq 5\,\mathbf{u}$ **then**
  $p \leftarrow a$;
else
  $l \leftarrow \text{round}(\log(c))$;
  $f \leftarrow n \oslash l$;
  $p \leftarrow a \otimes f$;
end

4.1 Experimental analysis of the Instability

Therefore, we revert to a by-hand, experimental analysis of expression (1). Extracting the value used when running the test case, one can let b vary in a small floating-point interval around a, for example of radius $600\,\mathbf{u}$, and study the errors made when computing $f(a,b)$ using an implementation similar to the one used in code_aster. In our case, we take

$$a = 4.2080034963016440 \times 10^{-5} \quad \text{and} \quad b \in \Big[a\,(1-600\,\mathbf{u}), a\,(1+600\,\mathbf{u})\Big]\,.$$

One can choose any language to perform such an analysis, as the underlying floating-point arithmetic will always be the same. For the present study, we chose to use Julia, which has the advantage of proposing an easy-to-use interval arithmetics library: `ValidatedNumerics` [18]. This library makes it very easy to compute precise values of $f(a,b)$, to be used as reference when evaluating errors produced by the tested implementation. More precisely, reference results in this study have been produced using an interval arithmetic based on an underlying 112-bit floating-point arithmetic. We can check *a posteriori* that both extremities of the resulting interval are rounded to the same double-precision floating-point value, which ensures that the 112-bit precision is enough to compute an accurate approximation of the real result. This value is then used as a reference to compute the relative error of the floating-point evaluation of $f(a,b)$.

As illustrated by the red square markers in Fig. 4, the initial implementation in code_aster produces very large relative errors, sometimes higher than 60 %. With such an implementation, evaluating $f(a,b)$ when b is very close to a (a few ulps) even produces infinite or NaN values.

4.2 Proposed Solution

In order to devise an alternative formulation of expression (1), equivalent in terms of real calculations but more stable with floating-point arithmetic, we need to make sure to avoid any catastrophic cancellation in the denominator. We therefore propose in this work to use the following re-definition of the function:

$$f(a,b) = \begin{cases} a & \text{if } a = b, \\ a\,\frac{\frac{b}{a}-1}{\log(\frac{b}{a})} & \text{otherwise.} \end{cases} \tag{2}$$

Furthermore, in order to evaluate this expression using a binary floating-point arithmetic, we propose to implement it using Alg. 1. In this algorithm, we denote by $\oplus$, $\ominus$, $\otimes$ and $\oslash$ the rounded-to-nearest floating-point versions of the basic operations. Notice that the test in Alg. 1 has been enlarged, so that future analyses using techniques like the one described in paragraph 3 are less likely to detect a (now hopefully fixed) unstable test.

The quality of this algorithm can be evaluated using the methodology presented in the previous section. Results shown by the blue x-shaped markers in Fig. 4 clearly illustrate the improvement brought by the use of the proposed algorithm: relative errors drop from the $[10^{-5}, 10^{-1}]$ range, down to 3 ulps.

4.3 Proof

In an industrial context, the previous study would probably be enough to validate the proposed formula: most engineers would accept such experimental results are a sufficient "guarantee" that the proposed expression is correct.

Nevertheless, for the sake of completeness, we present here a mathematical proof that the proposed formula actually provides good accuracy in all cases. Even if we depart from the pragmatic, industrial approach, such a proof will at least have a pedagogical interest, in that it gives a better understanding of why the proposed correction improves overall accuracy.

Theorem 1. *Let a and b be two floating point numbers. Then, assuming that no denormalized numbers appear during calculations, the operations described in Alg. 1 produce a result close to $f(a, b)$, with a relative error bounded in the first order by 10 ulps.*

We assume in this proof that no overflow occurs. If we define

$$x = \frac{b}{a},$$

then we have

$$c = x\,(1 + \varepsilon_{\mathrm{c}}), \tag{3}$$

where ε_{c} is a relative error smaller than machine precision: $|\varepsilon_{\mathrm{c}}| \leq \mathbf{u}$.

In the following, we will adhere to the convention that all relative round-off errors are denoted by ε_v, where v is the name of the variable storing the result of the operation, as defined in Alg. 1. The proof is presented in its entirety in Appendix A, and we only give a sketch of it here. We start by defining two sub-cases, corresponding to the branches of the test.

Case 1: a almost equal to b. This case corresponds to lines 3–4 in Alg. 1. The full proof for this case is presented in Appendix A.1. It relies on the fact that, since $1 - x$ is the beginning of the Taylor development of $\log(x)$ in 1, the approximation

$$\frac{x - 1}{\log(x)} \simeq 1$$

is accurate when $x \simeq 1$.

Case 2: *a* "far from" *b*. This case corresponds to lines 5–8 in Alg. 1. Starting from (3) and following the next statements in the algorithm, we have

$$n = (c-1)\ (1+\varepsilon_{\mathrm{n}})\,, \tag{4}$$

$$l = \log(c)\ (1+\varepsilon_{\mathrm{l}})\,, \tag{5}$$

$$\begin{aligned} f &= \frac{n}{l}\ (1+\varepsilon_{\mathrm{f}}) \\ &= \frac{c-1}{\log(c)}\ \frac{(1+\varepsilon_{\mathrm{n}})(1+\varepsilon_{\mathrm{f}})}{1+\varepsilon_{\mathrm{l}}}, \end{aligned} \tag{6}$$

$$\begin{aligned} p &= a\,f\ (1+\varepsilon_{\mathrm{p}}) \\ &= a\,\frac{c-1}{\log(c)}\ \frac{(1+\varepsilon_{\mathrm{n}})\,(1+\varepsilon_{\mathrm{f}})\,(1+\varepsilon_{\mathrm{p}})}{1+\varepsilon_{\mathrm{l}}} \\ &= a\,\frac{x-1}{\log(x)}\ \underbrace{\frac{c-1}{x-1}\ \frac{\log(x)}{\log(c)}}_{E_1}\ \underbrace{\frac{(1+\varepsilon_{\mathrm{n}})\,(1+\varepsilon_{\mathrm{f}})\,(1+\varepsilon_{\mathrm{p}})}{1+\varepsilon_{\mathrm{l}}}}_{E_2}. \end{aligned} \tag{7}$$

We obtain in (7) that p is an approximation of the desired quantity, with a relative error given by $e = E_1\,E_2 - 1$. While it is clear that E_2 is bounded and close to unity, a more thorough analysis of E_1 needs to be performed.

Indeed, when x and c are close to 1, all four terms appearing in E_1 are small, and it is not clear that E_1 can be bounded. We will thus define two sub-cases: $x \in \left[\frac{1}{2}, 2\right]$ and $x \notin \left[\frac{1}{2}, 2\right]$.

Case 2a: $x \notin \left[\frac{1}{2}, 2\right]$. This case is rather straightforward, since all terms are far enough from 0 for relative errors to stay bounded. Appendix A.2 presents the proof for this case, partly automatized with the Gappa proof assistant [4].

Case 2b: $x \in \left[\frac{1}{2}, 2\right]$. This case is maybe the most interesting one, as it explains why the proposed expression is more stable than the initial one. The idea here is that, since $1-x$ is the beginning of the Taylor development of $\log(x)$ in 1, the function $\frac{x-1}{\log(x)}$ can not get too close to 0 in the considered interval. Moreover, its derivative can be bounded, so that the small error between x and c can not have too catastrophic consequences.

5 Checking the effectiveness of corrections

We can finally re-assess the numerical stability of code_aster in order to check the effectiveness of the corrections made so far.

Before using Verrou and following the protocol of Sect. 2, an important question which arises at this stage is the validity of Random Rounding Arithmetic to assess compensated algorithms such as the ones introduced to fix the problem detected in Sect. 3.1. Indeed, such algorithms were designed to work in nearest

Table 2: Analysis of numerical instabilities of test-case `sdnl112a`

Version	Status nearest	rnd_1	rnd_2	rnd_3	# common digits $C(\text{rnd}_1, \text{rnd}_2, \text{rnd}_3)$
Before correction	OK	KO	KO	KO	6 6 6 * 3 0
After correction	OK	KO	OK	OK	9 10 8 * 5 0

rounding mode, and could very well be incompatible with directed or random rounding. Pioneering work has been performed on this topic in [7], which shows that compensated summations and dot products will indeed work with random rounding. However, a broader study of various kinds of algorithms still needs to be performed.

Table 2 presents the results of a re-assessment of the numerical stability of test-case `sdnl112a`. The first line in this table comes from the initial tests presented in Table 1; the second line corresponds to the same test performed on the version of code_aster in which the instabilities uncovered in Sect. 3 were fixed.

We can observe that two random rounding runs now pass, and the test stability has been improved by 2 to 4 decimal digits depending on the result. The 6^{th} result has no digit in common between random-rounding runs because it is expected to be close to 0 (and the test suite uses an absolute error to validate it, so this result is not alarming). However, the corrected version still fails the test in one random rounding run. Investigations show that this is because of the 5^{th} result, which is not surprising since the numerical stability of this result is still relatively low, with only 5 decimal digits in common between random rounding runs. This means that some work still needs to be done in order to correct inaccuracies in this test case.

6 Conclusion and Perspectives

We have presented in this paper a complete workflow for the analysis and improvement of the numerical quality of a large industrial code, as exemplified by code_aster. An important conclusion to get from this work is that such an analysis, which would probably have been out of reach a few years ago, is now feasible. This paves the way for tighter integration of numerical verification tools within industrial Verification and Validation processes.

This work also illustrates the wide variety of techniques which can be used to perform such an analysis, and the complementarity between them:

- Random Rounding Arithmetic (RRA), as implemented in Verrou, has successfully helped to bring out the numerical instabilities in code_aster, and to locate their origin in the source code;
- compensated algorithms have allowed increasing the accuracy of large dot products in the code;

– higher precision computations (whose correctness can be guaranteed ny Interval Arithmetic) have allowed to experimentally analyze an inaccurate expression and check the validity of the proposed reformulation;
– a mathematical proof has confirmed this validity over the whole range of arguments, while giving bounds on the produced error;
– RRA can again be used to assess the effectiveness of the corrections applied to code_aster.

Nevertheless, more work still needs to be carried out on such topics. Several sources of errors have been detected in the study, but we further studied and proposed corrections for only two of them. A straightforward extension would be to try and correct other errors to further improve the overall stability of code_aster and make all its test cases reproducible and portable across architectures. Also, steps involving the precise localization of errors and the proposition of corrections could be further streamlined, either in Verrou or in tools like Herbgrind (localization) and Herbie (correction).

References

1. Code_Aster: Structures and thermomechanics analysis for studies and research. http://www.code-aster.org/
2. Benz, F., Hildebrandt, A., Hack, S.: A dynamic program analysis to find floating-point accuracy problems. In: 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 453–462. ACM, New York, NY, USA (Jun 2012)
3. Denis, C., de Oliveira Castro, P., Petit, E.: Verificarlo: checking floating point accuracy through Monte Carlo Arithmetic. In: 23rd IEEE Internatinal Symposium on Computer Arithmetic (ARITH'23) (2016)
4. de Dinechin, F., Lauter, C., Melquiond, G.: Certifying the floating-point implementation of an elementary function using Gappa. IEEE Transactions on Computers 60(2) (2011)
5. Févotte, F., Lathuilière, B.: VERROU: Assessing Floating-Point Accuracy Without Recompiling (Oct 2016), `https://hal.archives-ouvertes.fr/hal-01383417`
6. Févotte, F., Lathuilière, B.: VERROU: a CESTAC evaluation without recompilation. In: International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics (SCAN). Uppsala, Sweden (Sep 2016)
7. Graillat, S., Jézéquel, F., Picot, R.: Numerical Validation of Compensated Algorithms with Stochastic Arithmetic (Sep 2016), `https://hal.archives-ouvertes.fr/hal-01367769`
8. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008 pp. 1–70 (2008)
9. Jézéquel, F., Chesneaux, J.M., Lamotte, J.L.: A new version of the CADNA library for estimating round-off error propagation in Fortran programs. Computer Physics Communications 181(11), 1927–1928 (2010)
10. Lam, M.O., Hollingsworth, J.K., Stewart, G.: Dynamic floating-point cancellation detection. Parallel Computing 39(3), 146–155 (2013)
11. Lamotte, J.L., Chesneaux, J.M., Jézéquel, F.: CADNA_C: A version of CADNA for use with C or C++ programs. Computer Physics Communications 181(11), 1925–1926 (2010)

12. Montan, S.: Sur la validation numérique des codes de calcul industriels. Ph.D. thesis, Université Pierre et Marie Curie (Paris 6), France (2013), in French
13. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI) (2007)
14. Neumaier, A.: Rundungsfehleranalyse einiger verfahren zur summation endlicher summen. ZAMM (Zeitschrift für Angewandte Mathematik und Mechanik) 54, 39–51 (1974)
15. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM J. Sci. Comput. 26 (2005)
16. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatloc, Z.: Automatically improving accuracy for floating point expressions. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15). Portland, Oregon, USA (Jun 2015)
17. Sanchez-Stern, A., Panchekha, P., Lerner, S., Tatlock, Z.: Finding root causes of floating point error with herbgrind, `arXiv:1705.10416v1 [cs.PL]`
18. Sanders, D.P., Benet, L., Kryukov, N.: The julia package `ValidatedNumerics.jl` and its application to the rigorous characterization of open billiard models. In: International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics (SCAN). Uppsala, Sweden (Sep 2016)
19. Sterbenz, P.H.: Floating Point Computation. Prentice-Hall, Englewood Cliffs, NJ (1974)
20. Stott Parker, D.: Monte Carlo arithmetic: exploiting randomness in floating-point arithmetic. Tech. Rep. CSD-970002, University of California, Los Angeles (1997)
21. Vignes, J.: A stochastic arithmetic for reliable scientific computation. Mathematics and Computers in Simulation 35, 233–261 (1993)
22. Zeller, A.: Why Programs Fail. Morgan Kaufmann, Boston, second edn. (2009)

A Complete proof

A.1 Case 1: a almost equal to b

We treat in a first step the case where the condition in the "if" statement at line 3 of Alg. 1 applies. In this case, a and b are close enough to one another for Sterbenz lemma [19] to hold, which means that no additional error is made when computing n:

$$n = c - 1 = x\,(1 + \varepsilon_{\mathrm{c}}) - 1.$$

The condition tested at the beginning of Alg. 1 therefore implies that:

$$\frac{1 - 5\,\mathbf{u}}{1 + \mathbf{u}} \leq x \leq \frac{1 + 5\,\mathbf{u}}{1 - \mathbf{u}},$$

and

$$\frac{-6\,\mathbf{u}}{1 + \mathbf{u}} \leq x - 1 \leq \frac{6\,\mathbf{u}}{1 - \mathbf{u}}. \tag{8}$$

The algorithm returns a in this case, instead of the exact value

$$f(a,b) = a\,\frac{x-1}{\log(x)},$$

so that the relative error is given by:

$$\begin{aligned}
e_0 &= \frac{a - f(a,b)}{f(a,b)} \\
&= \frac{\log(x)}{x-1} - 1 \\
&= \frac{\log(1+\epsilon)}{\epsilon} - 1 && \text{(where } \epsilon = x-1\text{)} \\
&= \frac{1}{\epsilon}\left(\sum_{n=0}^{\infty} \frac{(-1)^n\,\epsilon^{n+1}}{n+1}\right) - 1 && \text{(Taylor expansion of the log function)} \\
&= \sum_{n=1}^{\infty} \frac{(-1)^n\,\epsilon^n}{n+1}. && (9)
\end{aligned}$$

Assuming $\epsilon \geq 0$, we have

$$\forall n \in \mathbb{N}, \qquad \frac{\epsilon^n}{n+1} > \frac{\epsilon^{n+1}}{n+2},$$

so that grouping terms in pairs in (9) yields

$$e_0 = -\sum_{k=1}^{\infty}\left[\frac{\epsilon^{2k-1}}{2k} - \frac{\epsilon^{2k}}{2k+1}\right] \leq 0$$

and

$$e_0 = -\frac{\epsilon}{2} + \sum_{k=1}^{\infty}\left[\frac{\epsilon^{2k}}{2k+1} - \frac{\epsilon^{2k+1}}{2k+2}\right] \geq -\frac{\epsilon}{2}.$$

The case where $\epsilon < 0$ is treated similarly, so that we get

$$|e_0| \leq \frac{|\epsilon|}{2} \leq \frac{3\,\mathbf{u}}{1-\mathbf{u}},$$

where we injected (8) in the last inequality. This last result shows that in this case, the relative error is bound by 3 ulps in the first order.

A.2 Case 2a: $x \notin \left[\frac{1}{2}, 2\right]$

We assume in this case that $x \notin \left[\frac{1}{2}, 2\right]$, and will focus on the sub-case where $x > 2$ (the other subcase, $x < \frac{1}{2}$, can be handled in a similar way).

Starting from (3), and knowing that the logarithm is a monotonically increasing function, we have:

$$\log(c) = \log(x\,(1+\varepsilon_c)) = \log(x) + \log(1+\varepsilon_c),$$
$$\implies |\log(c) - \log(x)| \leq \log(1+\mathbf{u}) \leq \mathbf{u},$$

where the last inequality was obtained by noting that the logarithm is convex, and its derivative in 1 is 1. A rather simple Gappa script, presented in Fig. 5 can prove the rest. In this script, all capital letters are ideal, real values corresponding to the approximations computed in Alg. 1 and represented by lower-case letters. We denote `LX` $= \log(x)$, and `LE` $= \log(c) - \log(x)$. The bound on `LE` used as hypothesis comes from the simple computation above, the bound on `LX` are those of the logarithm over the range of double-precision floating-point numbers. Other bounds come from double-precision floating-point limits.

Gappa can prove that the relative error produced by Alg 1 in this case is bounded by approximately 8.9×10^{-16}, which is compatible with the bounds stated in Theorem 1. It should be noted however that Gappa can't validate this script for too small values of a, probably denoting a problem with denormalized values.

```
# a and b are double-precision floating-point values
@rnd = float<ieee_64, ne>;
a = rnd(A);
b = rnd(B);

# Real computation (log(X) = LX)
X = b / a;
F = (X - 1) / (LX + 0);
P = a * F;

# FP computation   (log(c) = LX + LE)
c = rnd(X);
n rnd= c - 1;
l rnd= LX + LE;
f rnd= n / l;
p rnd= a * f;

{
  # Hypotheses
  (  a    in [1b-1000, 1.8e308]  # upper bound coming from
  /\ X    in [2,       1.8e308]  # binary64 limits
  /\ LX   in [0.5,     710]
  /\ |LE| <= 1b-53)

  # Conclusion
  -> p -/ P in ?
}
```

```
Results:
    p -/ P in [-1152921504606846483b-110 {-8.88178e-16, -2^(-50)},
               1152921504606846483b-110 { 8.88178e-16,  2^(-50)}]
```

Fig. 5: Gappa script used to prove case 2a

A.3 Case 2b: $x \in \left[\frac{1}{2}, 2\right]$

We finally study here the case when a and b are close to one another: $x \in \left[\frac{1}{2}, 2\right]$. Let us define

$$g(x) = \frac{\log(x)}{x-1},$$

so that, recalling the expression of E_1 from (7),

$$E_1 = \frac{g(x)}{g(c)} = \frac{g(x)}{g(x + x\,\varepsilon_c)}.$$

We have:

$$\begin{aligned} |g(x + x\,\varepsilon_c) - g(x)| &\leq x\,|\varepsilon_c| \sup_{y \in [x, x + x\,\varepsilon_c]} |g'(y)| \\ &\leq x\,|\varepsilon_c| \sup_{y \in [\frac{1-\mathbf{u}}{2}, 2+2\mathbf{u}]} |g'(y)| \\ &\leq 0.6\,|\varepsilon_c|, \end{aligned}$$

where the last inequality was obtained by noticing that

$$\forall y \in \left[\frac{1-\mathbf{u}}{2}, 2 + 2\mathbf{u}\right], \quad g'(y) \in [-0.3, -0.1],$$

as shown by a simple interval analysis. A similar interval analysis shows that

$$\forall y \in \left[\frac{1}{2}, 2\right], \quad g(y) \geq \frac{1}{2},$$

so that

$$\left|\frac{g(x + x\,\varepsilon_c) - g(x)}{g(x)}\right| \leq 1.2\,|\varepsilon_c|,$$

and thus, recalling the expression of E_1 from (7),

$$\frac{1}{1 + 1.2\,|\varepsilon_c|} \leq E_1 \leq \frac{1}{1 - 1.2\,|\varepsilon_c|}.$$

Putting all previous results together, we therefore have

$$\frac{(1-\mathbf{u})^3}{(1 + 1.2\,\mathbf{u})\,(1+\mathbf{u})} \leq 1 + e \leq \frac{(1+\mathbf{u})^3}{(1 - 1.2\,\mathbf{u})\,(1-\mathbf{u})},$$

which proves that, in the first order, the relative error in this case is bounded by 6 ulps. It is interesting to note here that, depending on the specific floating-point implementation of the logarithm, l might not be correctly rounded and error term ε_1 might be bounded by several ulps. Should this happen, the relative error on the result of Alg. 1 would be higher, but still bounded.