VERROU: Assessing Floating-Point Accuracy Without Recompiling

François Févotte, Bruno Lathuilière

VERROU: Assessing Floating-Point Accuracy Without Recompiling

François Févotte and Bruno Lathuilière EDF R&D
1 av. Général de Gaulle
92140 Clamart, FRANCE
Email: {francois.fevotte, bruno.lathuiliere}@edf.fr

October 18, 2016

Abstract

As an industrial facility relying on numerical simulation to improve the safety and efficiency of its electricity production units, EDF is committed to ensure that all the numerical simulation codes it develops and uses are correctly validated and verified. Within this context, the accuracy of floating-point operations has progressively become one of the important topics to study, especially since computing codes are exploited on ever more powerful hardware to solve ever larger problems. The Verification and Validation (V&V) process should therefore include the monitoring of inaccuracies introduced by floating-point arithmetic, as well as the verification that they are kept within acceptable limits. In this paper, we present the VERROU tool, a valgrind-based system which uses Monte Carlo Arithmetic (MCA) to monitor the accuracy of floating-point operations without needing to instrument the source code or even recompile it. This tool is therefore well-suited to be part of an industrial V&V process. It has been successfully tested both on small-scale, well understood numerical applications, and on large-scale, more complex industrial computing codes.

Keywords: Monte Carlo Arithmetic, floating-point arithmetic, numerical verification, round-off errors

1 Introduction

As numerical simulation tools take an ever more important place within industrial processes, Verification & Validation (V&V) has become a major concern both for code developers and users. Within this context, and seeing the large amount of work devoted to numerical verification in the academic world, it might come as a surprise that floating-point accuracy assessment only recently started making its way to the industry. Indeed, most simulation codes manipulate floating-point values and perform computations on them using the rules promoted by standard IEEE-754 [1]. As an ever growing number of such computations are routinely performed, rounding errors have become one of the key sources of inaccuracies in the results, along with discretization and iterative convergence issues. which were until then the only focus of verification studies.

Furthermore, numerous tools exist to diagnose floating-point problems, among which CADNA [2] is one of the most advanced. It allows instrumenting the source code, replacing standard floating-point arithmetic by Discrete Stochastic Arithmetic (DSA), in order to assess floating-point inaccuracies, detect their origin in the source code, and follow their propagation throughout the computation. CADNA has already been successfully used on large industrial simulation codes [3], which makes it a natural point of comparison throughout this paper.

Other tools exist, such as FpDebug [4], which performs the Dynamic Binary Instrumentation (DBI) of an input executable, in order to compare each floating-point value in the program with one obtained with higher precision. FpDebug outputs a graph indicating, for each floating-point operation, whether inaccuracies come from input data or from the operation itself.

A third tool worth mentioning here is Craft HPC [5], another tool performing DBI in order to detect cancellation errors, and furthermore optimize the use of single- and double-precision variables throughout the source code in order to balance speed and accuracy.

However, despite the availability of these tools and the growing part taken by round-off errors in results inaccuracies, only a few industrial V&V processes include numerical verification of simulation codes. A few reasons might be proposed to explain this observation: first, some development teams are not aware of the ubiquity of rounding errors. Some, on the contrary, are convinced that floating-point inaccuracies are a fatality against which nothing can be done. Second, and more importantly, analyzing an industrial code is a far more difficult task, when compared to smaller programs:

- instrumenting the source code with CADNA can be hard, and there generally are numerous dependencies to third-party software libraries of which the development team only has limited understanding.
- Industrial simulation codes routinely produce gigabytes of results, an in-depth analysis of which is in itself often a challenge. In this context, the size of graphs output by FpDebug or of lists of cancellations detected by Craft HPC can quickly become overwhelming.

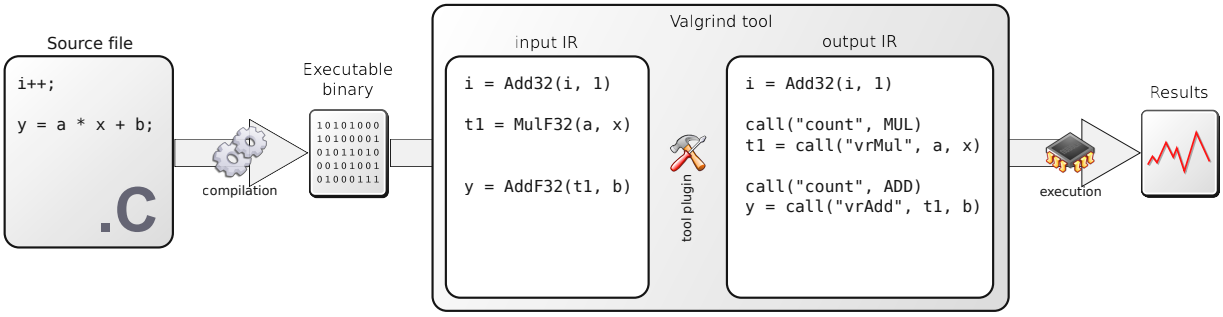


Figure 1: Schematic description of dynamic binary analysis with Valgrind

As a consequence, few developers are willing to initiate such work before being certain that the code quality will improve.

In this paper, we introduce VERROU,[1] a tool attempting to bridge the gap between simulation code developers and in-depth numerical verification tools. As such, VERROU aims at providing the following features:

- Low entry cost: no need for source code instrumentation, nor even recompilation [6]. To this end, VERROU performs DBI using the Valgrind framework.
- Reliable detection of floating-point problems, using Monte Carlo Arithmetic (MCA).
- Easy and seamless introduction within industrial V&V processes to analyze floating-point errors.
- As low as possible run-time overhead.

The rest of this paper is organized as follows. In part 2, we briefly describe the principles of MCA, which VERROU uses. The implementation of the tool is presented in part 3, where we first review the Valgrind Dynamic Binary Instrumentation framework, before showing how it is used to implement MCA. Finally, part 4 is devoted to experimental tests of VERROU. Small-scale, well understood numerical applications allow validating the implementation of MCA, while the study of a large-scale, more complex industrial code showcases how VERROU meets its objectives of integrating into an industrial V&V process.

2 Monte Carlo Arithmetic

Monte Carlo Arithmetic (MCA) is a floating-point arithmetic system in which inexact values and round-off errors are respectively modeled with randomization of the input values and the output results. Systematically replacing all floating-point operations in a program with their respective MCA counterparts therefore turns any set of program executions into a sample of a random variable representing the result and its floating-point accuracy.

In this section, we try to briefly describe the parts of MCA that are relevant to VERROU. The interested reader can refer to [7] for the details.

2.1 Inexact values

The MCA approach proposes to model inaccurate values as random variables. If $\tilde{x}$ is an inaccurate value approaching value x to s significant digits in base β, then $\tilde{x}$ can be represented as the random variable

$$\tilde{x} = x + \beta^{e-s}\,\xi,$$

where e is the order of magnitude of x in base β, and ξ is a random variable on the interval $[-\frac{1}{2}, \frac{1}{2}]$.

Conversely, knowing such a random variable, it is possible to retrieve the number s of exact significant digits of the inaccurate value it represents as

$$s = \log_\beta \left(\frac{\mathrm{E}(\tilde{x})}{\mathrm{S}(\tilde{x})} \right),$$

where $\mathrm{E}(\tilde{x})$ and $\mathrm{S}(\tilde{x})$ respectively denote the expected value and the standard deviation of random variable $\tilde{x}$.

Therefore, assuming that repeated executions of a program under MCA produce a sample of the (inexact) result expressed as such a random variable, it is possible to perform a statistical analysis to compute estimates $\hat{\mu}$ and $\hat{\sigma}$ of

[1] The name VERROU was chosen as a tribute to CADNA, since *cadenas* and *verrou* are two French words respectively translating to "padlock" and "bolt".

$\mathrm{E}(\tilde{x})$ and $\mathrm{S}(\tilde{x})$. From there, $\hat{\mu}$ can be considered to represent the computation result with a number of significant digits approximately given by

$$s \simeq \log_\beta \left(\frac{\hat{\mu}}{\hat{\sigma}} \right). \tag{1}$$

In the following, the term "*inexact value*" will refer to this definition of a random variable modelling a value and the inaccuracies attached to it.

2.2 Assessment of inaccuracies due to floating-point arithmetic

We now describe the Monte Carlo arithmetic which can be used by a program to output *inexact values* modelling the inaccuracies resulting from floating-point arithmetic. In what follows, we will suppose that all floating-point values are defined using a given, known precision (which in practice will be either the single or double binary precision of IEEE-754).

The full MCA approach can be summarized as follows: if x and y are floating point values and $*$ is a binary operator, we introduce the MCA floating-point implementation of $*$, denoted by $\circledast$ and defined by

$$x \circledast y = \mathsf{random\_round}\left(\mathsf{value}(x) * \mathsf{value}(y)\right),$$

where value is a function which maps values to *inexact values*, and is defined as

$$\mathsf{value}(v) = \begin{cases} v & \text{if } v \text{ can be expressed exactly,} \\ \mathsf{inexact}(v) & \text{otherwise,} \end{cases}$$

and $\mathsf{inexact}$ is a function randomizing value v. The function $\mathsf{random\_round}$ computes a random rounding of a given value:

$$\mathsf{random\_round} = \mathsf{round} \circ \mathsf{value}.$$

Using $\mathsf{inexact}$ in the expressions above makes the result of an MCA operation a random variable, unless operands are exact values and the result can itself be represented exactly with the considered precision. Moreover, a careful choice of the definition of $\mathsf{inexact}$ can lead the output result to be an *inexact value* in the sense of section 2.1.

Calling value on inputs x and y is known as "input precision bounding", and helps detecting catastrophic cancellations by introducing random lower-order digits which will track the loss of significance resulting from cancellation.

On the other hand, calling $\mathsf{random\_round}$ on the result of the operation is known as "random rounding", and models forward errors. VERROU does not implement the full MCA approach: it only considers random rounding, which is thus described in more details in the following.

For any value v, $\mathsf{random\_round}(v)$ can have different definitions (corresponding to different definitions of $\mathsf{inexact}$), including:

- `round_nearest`: rounds v to the nearest exactly representable value in the given precision. In this case, $\mathsf{random\_round}$ is identical to the ordinary round function.
- `round_random`: rounds v up or down with probability $\frac{1}{2}$. It is similar to the random rounding mode used by the CESTAC method upon which CADNA is built.
- `round_average`: rounds v up or down, with probabilities proportional to the distance from v to either of these two alternatives. This type of rounding, initially introduced and studied in [8], is also referred to as "`uniform_absolute` output randomization". As shown in [7], this type of random rounding produces an *inexact value* with zero expected bias, *i.e.* it eliminates the expected error from rounding.

3 VERROU: floating-point accuracy assessment

We now describe the implementation of VERROU. In particular, we briefly introduce Valgrind, which is used to perform Dynamic Binary Instrumentation. We then describe how MCA has been implemented within this framework. In a third part, we detail the use of VERROU for numerical verification, and particularly the ways it can be introduced into an industrial V&V process. Finally, we describe more advanced instrumentation features implemented in VERROU, as well as the limitations which still remain to overcome.

3.1 Dynamic binary analysis based on Valgrind

In communities of developers, Valgrind[2] is a well-known tool used mostly for memory debugging. Less known is the fact that it is actually a whole framework allowing to build Dynamic Binary Instrumentation (DBI) tools [9]. As the tool is already heavily used in numerous development and V&V toolchains, it was a logical choice for the base component upon which to build VERROU.

Describing the full internal workings of Valgrind is well outside the scope of this paper. In this section, we will try to briefly describe how DBI works from the perspective of a Valgrind tool writer and user. Figure 1 presents a schematic overview of Valgrind's dynamic binary re-compilation process. On the left part of the figure is an example C source file, which has been compiled to produce a binary executable file. This will be the analyzed program. A Valgrind tool, represented by the large central box, is composed of the Valgrind core – which implements all the heavy DBI operations – and a tool plugin – which is the only part left for the tool developer to implement. For example, Valgrind's default and most popular tool is Memcheck,[3] a memory error detector which is often confused with Valgrind itself.

Valgrind uses a *disassemble-and-resynthetize* strategy, which means that the first task performed by the core consists in converting machine code from the instrumented binary to an Intermediate Representation (IR, represented as the left column and labelled "input" in the Valgrind box on figure 1). The tool plugin is then responsible to transform the original IR into an instrumented IR (labeled "output" on the figure). In the end, the IR is converted back to machine code to be executed.

In the example given on figure 1, the first line in the C code is converted to an integer instruction in the IR, which is left untouched by the tool plugin (it is copied to the output IR). The second line in the C code is converted to two floating-point instructions in the IR. Each of these instructions will be erased, and replaced by other instructions in the output IR. In this example, custom functions are called to count instrumented floating-point instructions (pure side effect), and replace the original instruction with another implementation (which means putting the desired result back where it would have been).

3.2 Implementation of MCA operations

VERROU is built as a Valgrind tool, which allows forgetting details about DBI and focusing instead on the implementation of MCA operations. However, developing within Valgrind has its own constraints, most of which are related to the fact that the Valgrind code will be executed in the same process as the instrumented code. In particular, two main limitations are impacting the development of VERROU:

1. No calls to external libraries are allowed (even the C standard library is not available; the Valgrind core proposes its own implementation of a subset of the C library). This makes it difficult to re-use available MCA implementations such as MCALIB [10], since they would need to be made compatible with Valgrind's constraints, which in turn would require adapting all their dependencies to Valgrind. Among these dependencies is for example MPFR [11], whose adaptation is feasible [4] but extremely heavy.

2. As for floating-point operations, only rounding to the nearest is supported. This prevents changing the CPU rounding mode to easily implement the `round_random` output randomization as proposed in [12].

MCA operations in VERROU are therefore reimplemented from scratch, using as few dependencies as possible. The approach retained is based on error-free transformations (EFT). Following the notations and definitions of section 2, given two floating-point operands x and y, the objective is to compute

$$x \circledast y = \mathsf{random\_round}(x * y),$$

using only rounded to nearest floating point operations. Supposing that there exists an error-free transformation for binary operator $*$, one can write

$$x * y = \mathsf{round}(x * y) + \delta, \tag{2}$$

where $\mathsf{round}(x * y)$ is the rounded result of the floating-point operation, and δ is a floating-point value which can be computed exactly using standard floating-point arithmetic. In the following, let us denote by z the exact operation result, and $\lfloor z \rfloor$, $\tilde{z}$ and $\lceil z \rceil$ its approximate floating-point values respectively rounded downwards, to nearest, and upwards as defined by the IEEE-754 standard. Knowing $\tilde{z}$, the sign of δ allows determining the downwards and upwards-rounded values of z:

$$\begin{array}{ll} \text{if } \delta > 0, & \text{if } \delta < 0, \\ \quad \left| \begin{array}{l} \lfloor z \rfloor = \tilde{z}, \\ \lceil z \rceil = \tilde{z} + \mathrm{ulp}(\tilde{z}), \end{array} \right. & \quad \left| \begin{array}{l} \lfloor z \rfloor = \tilde{z} - \mathrm{ulp}(\tilde{z}), \\ \lceil z \rceil = \tilde{z}, \end{array} \right. \end{array}$$

[2]Valgrind: `http://www.valgrind.org/`
[3]Memcheck: `http://valgrind.org/docs/manual/mc-manual.html`

where $\mathrm{ulp}(\tilde{z})$ represents the difference between two consecutive representable floating-point numbers for values of the order of magnitude of $\tilde{z}$ with the considered precision.

The choice of an error-free transformation in equation (2) must be made in accordance with the transformed operator, and should ideally be as cost-effective as possible. In VERROU, Knuth's `twosum` algorithm [13] is used for the sum and subtraction operations, and Dekker's `twoprod` [14] for the product. The case of the division is a bit more difficult to handle and is developed in further details in the next section.

The MCA result is then computed as follows: if $\delta = 0$, then $\tilde{z} = z$ is returned. Otherwise, the result is chosen according to a user-determined strategy:

- `NEAREST`, `DOWNWARD`, `UPWARD`: the relevant IEEE-754 rounding mode is emulated, and $\tilde{z}$, $\lfloor z \rfloor$ or $\lceil z \rceil$ is returned accordingly.
- `RANDOM`: this mode corresponds to the `round_random` strategy presented above. $\lfloor z \rfloor$ or $\lceil z \rceil$ is returned randomly, with equiprobability.
- `AVERAGE`: this mode corresponds to the `round_average` strategy presented above. $\lfloor z \rfloor$ or $\lceil z \rceil$ is returned randomly, with probabilities given by:

$$\text{if } \delta > 0, \quad \begin{cases} p(\lfloor z \rfloor) = 1 - \dfrac{\delta}{\mathrm{ulp}(\tilde{z})}, \\ p(\lceil z \rceil) = \dfrac{\delta}{\mathrm{ulp}(\tilde{z})}, \end{cases} \qquad \text{if } \delta < 0, \quad \begin{cases} p(\lfloor z \rfloor) = \dfrac{\delta}{\mathrm{ulp}(\tilde{z})}, \\ p(\lceil z \rceil) = 1 - \dfrac{\delta}{\mathrm{ulp}(\tilde{z})}. \end{cases}$$

3.3 Implementation of the division

If a and b are two floating-point numbers, we try to develop in this section a transformation without too much error for the division operation:

$$\frac{a}{b} \simeq q + r, \tag{3}$$

with

$$q = \mathsf{round}(a/b).$$

Our objective here is thus to evaluate, at least approximately, the remainder r, so that it can be used to compute an MCA version of the division operation, in the same way as value δ in the previous section. In order to evaluate an approximation to r, we propose algorithm 1.

Input: a, b
Output: $\tilde{r}$ such that $a/b \simeq \mathsf{round}(a/b) + \tilde{r}$.
1 $q \leftarrow \mathsf{round}(a/b)$;
2 $(p, s) \leftarrow \mathsf{twoprod}(b, q)$;
3 $t \leftarrow \mathsf{round}(a - p)$;
4 $u \leftarrow \mathsf{round}(t - s)$;
5 $\tilde{r} \leftarrow \mathsf{round}(u/b)$;

Algorithm 1: Approximate evaluation of the remainder in a floating-point division.

Multiplying equation (3) by b yields

$$a = b\,q + b\,r.$$

An algorithm like `twoprod` allows computing an EFT for the product $b\,q$:

$$b\,q = p + s,$$

where

$$p = \mathsf{round}(b\,q).$$

Thus, we obtain

$$b\,r = a - p - s. \tag{4}$$

Let us try and compute $b\,r$. Since floating-point numbers q and p are computed with exact rounding,

$$\begin{aligned} q &= \frac{a}{b}\,(1+\epsilon_1) && \text{where } |\epsilon_1| < \epsilon_\mathrm{m}, \\ p &= b\,q\;(1+\epsilon_2) && \text{where } |\epsilon_2| < \epsilon_\mathrm{m}, \end{aligned}$$

where ϵ_m is machine precision. This implies

$$p = a\,(1+\epsilon_1)\,(1+\epsilon_2)\,,$$

and thus

$$\frac{p}{2} \leqslant a \leqslant 2\,p.$$

From Sterbenz' lemma, $a - p$ is a floating-point number, so that

$$\begin{aligned} t &\coloneqq \mathsf{round}(a-p) \\ &= a - p. \end{aligned}$$

Injecting this result in equation (4) yields $b\,r = t - s$, which will then be computed with exact rounding:

$$\begin{aligned} u &\coloneqq \mathsf{round}(t-s) \\ &= \mathsf{round}(b\,r) \\ &= b\,r\,(1+\epsilon_3) && \text{where } |\epsilon_3| < \epsilon_\mathrm{m}, \end{aligned}$$

which in turns yields

$$\begin{aligned} \tilde{r} &\coloneqq \mathsf{round}(u/b) \\ &= \frac{u}{b}\,(1+\epsilon_4) && \text{where } |\epsilon_4| < \epsilon_\mathrm{m}, \\ &= r\,(1+\epsilon_3)\,(1+\epsilon_4)\,. \end{aligned}$$

3.4 Use and introduction in an industrial V&V process

Using Valgrind's tooling makes it very easy to run any executable binary within an MCA context using VERROU. The command line only needs a slight modification:

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

When run within this MCA context, the instrumented program executes in a normal way, except output results are *inexact values* samples. VERROU also appends its own output, indicating how many floating-point instructions were detected and successfully instrumented.

As underlined in the introduction, it is of crucial importance that VERROU be compatible with the strong requirements of industrial V&V processes.

Figure 2 presents a schematic view of non-regression testing within a typical industrial Verification & Validation process. The code is compiled from sources, then run on any number of test cases to produce results. Non-regression testing tools are then used to compare these results with previously stored reference results, and produce discrepancy measurements.

As seen above, in order to perform numerical validation of a code using VERROU, two major steps are needed:

1. run the code multiple times under MCA using VERROU;
2. perform a statistical analysis of the results sample to produce accuracy estimates.

VERROU itself allows easily tackling the problem of running a program under MCA. However, it does not help with the statistical analysis, which is entirely left for the user to implement. While this might seem to be a simple enough task for small-scale codes, it is often not so easy in the case of large-scale, industrial codes, who routinely produce several gigabytes of results data on each run.

Fortunately, large-scale, industrial codes often have non-regression testing suites as described above. And it is precisely the aim of such tools to be able to compare several executions of the computing code and produce discrepancy measurements. Figure 3 presents two ways of introducing VERROU into a non-regression testing process to allow for numerical verification. On the left (figure 3a), the regression test suite is simply run in the execution context of VERROU, so that results become MCA *inexact values*. Non-regression testing tools are then used as usual to compare

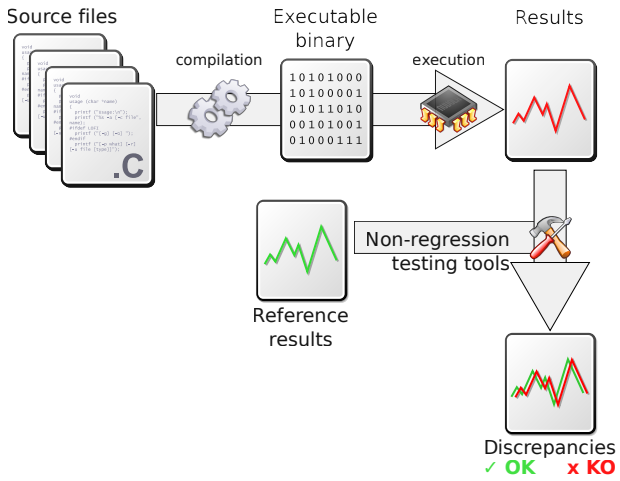
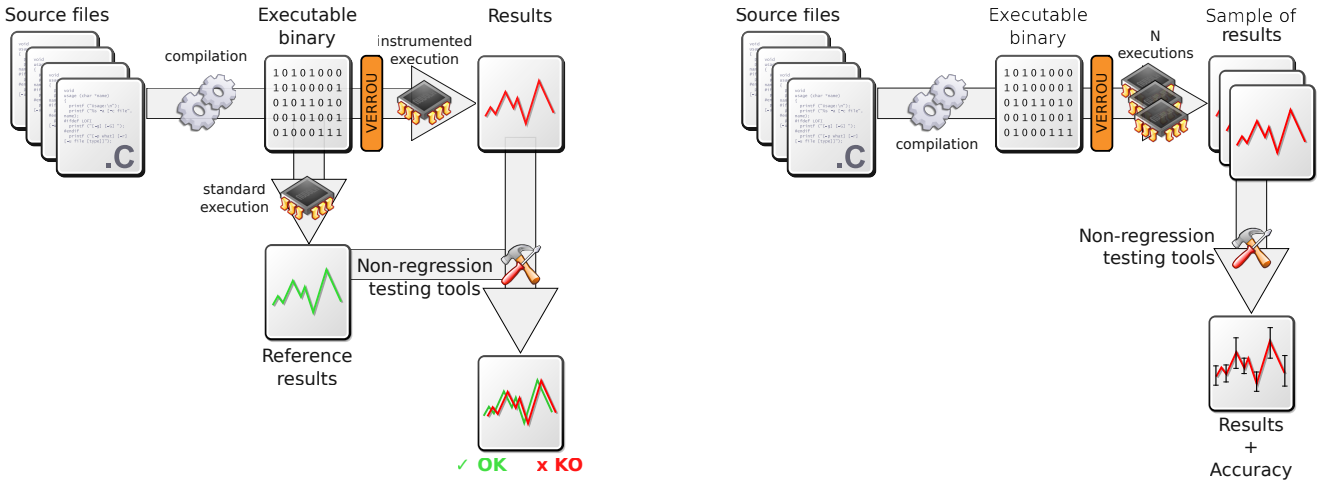


Figure 2: Schematic view of an industrial non-regression testing process.



(a) No additional work needed

(b) Light modifications of the non-regression testing tools

Figure 3: Introduction of VERROU into an industrial Verification & Validation process: non-regression testing tools can be reused for numerical verification.

perturbed results to reference results which are obtained through a standard execution of the code. The measured discrepancies give an indication as to whether accumulated errors in each test case of the suite are acceptable. The major advantage of such a scenario is its ease of implementation: no modification is needed. However, its main drawback is the lack of statistics. This can be mitigated by running the test suite several times, or relying on the fact that the test suite will be run every day or so during all the code development phase.

Figure 3b presents a slightly more complex way of introducing VERROU in an industrial non-regression testing process. In this case, each test case is run several times within the MCA execution context provided by VERROU. Non-regression testing tools need to be modified to compare all MCA results between them and produce statistics, instead of relying on a single reference result. This produces more reliable estimates for the accuracy of the results.

3.5 Advanced instrumentation features

The main advantage of Valgrind and VERROU lies in the fact that no source modification is required. This, however, comes with a downside: it is difficult to relate an unexpected behavior to its origins in the source code (when they are available). To partially mitigate this problem, we use Valgrind's client request mechanism: the user can annotate parts of the code to generate dummy instructions in the executable. These instructions will then be interpreted by VERROU during the run, but have no effect when the binary is executed outside Valgrind.

A first feature taking advantage of this mechanism is the possibility to activate or deactivate MCA to delimit sections in the program which are run in standard floating-point arithmetic. This feature is very useful in practice, since it allows roughly locating instability sources by trial and error. Another benefit offered by this feature is to avoid instabilities which are not caused by round-off errors *per se*, but rather by the stochastic approach itself. One such problem arises when a pure function is called twice with the same data input. Under standard floating-point arithmetic, whatever round-off errors happen in the code, the same result is always computed, which is a property on which developers might rely. However, determinism disappears under MCA, which may lead to incorrect program execution.

Rewriting programs so that they don't rely on determinism is not always a good option in terms of performance. On the other hand, although locally deactivating MCA restores determinism, it also prevents real instabilities in the function to be detected. Therefore, in order to correctly account for deterministic constraints, another client request in VERROU allows annotating functions so that they compute deterministic results, but still propagate round-off errors using MCA. This feature is implemented by seeding the random generator in a deterministic way on entry of such functions.

3.6 Delta-debugging and localization of errors

The features mentioned above allow to implement coarse-grained localization of the sources of floating-point errors by performing a bisection on the portions of the binary which are instrumented. Such a search is best conducted using specialized procedures like the Delta-Debugging algorithm [15, 16, 17].

VERROU comes with a utility tool, called VERROU-DD, which simplifies the localization of errors by automating the following process:

1. A first run of the test case is performed, without instrumentation. Results of this run are considered valid. A list of all functions (or rather, symbols) encountered during the run is generated. This list will become the list of code parts to check for floating-point errors (so-called "deltas", in the Delta-Debugging terminology).

2. The DD_max variant of the Delta-Debugging algorithm is performed on the list of symbols. The output is a maximal set of failure-inducing symbols, in the sense that perturbating floating-point rounding-modes in the instructions of each of these symbols produces failing results. Conversely, perturbating floating-point rounding modes in all instructions of the symbols not listed in the DD_max set is guaranteed to yield a passing result.

3. A second Delta-Debugging pass is performed at the grain-size of source code lines: for each symbol listed in the DD_max set, the following steps are performed:

   (a) A first reference run is performed to generate the list of source code lines encountered in the symbol.

   (b) The DD_max algorithm is performed on the list of source code lines. As for the first pass, this generates a maximal set of failure-inducing source code lines.

3.7 Limitations of VERROU

Calling a mathematical function such as cos is either translated to a specific assembly operation, or is delegated to a software library. In the former case, VERROU sees the special instruction, which it does not handle yet. MCA is not applied in this case. In the latter case, all VERROU sees is a series of floating-point computations implemented by the mathematics library. However, these algorithms have been designed only for the nearest rounding mode and are not compatible with MCA. For example, with random rounding mode in double precision, cos may return a value greater than 42. Implementing a way to automatically circumvent this problem is one of the most important perspectives for VERROU. But for now, users have to use client requests to deactivate MCA during each call to a trigonometric function.

For each floating point operation (+,-,*,/), three types of instructions can be found in an executable binary file:

1. scalar instructions of the SSE set;

2. SIMD instructions of the SSE and AVX sets;

3. scalar instructions of the x87 set.

SSE scalar instructions are fully instrumented by VERROU. However, only double precision vector instructions of the SSE set are currently handled. These limitations should be removed in the short term.

Valgrind –and consequently VERROU– is unable to properly instrument x87 scalar instructions. Nowadays, these instructions are no more generated by the compiler,[4] except for 80-bit instructions,[5] which are used for `long double` variables. With VERROU, a `long double` is treated as a `double` (but instructions are counted separately, so that the user is able to know exactly what has been instrumented).

4 Experimental evaluation

In this part, we evaluate VERROU in various contexts. Well-understood numerical benchmarks are first used to validate our implementation of MCA. In a second step, we present the use of VERROU in the real-world numerical verification of an industrial simulation code.

[4] unless explicitly asked using switches such as `-mfpmath=387`

[5] It should be noted that double rounding problems can occur with 80-bit instructions, due to the different size of the mantissa in the register and in memory.

Computation method	r_1	r_2
exact (rounded)	0.6062438663	0.6053616575
IEEE-754 (`float`, nearest)	0.6061973	0.6054083
error	*0.0000466*	*0.0000466*
VERROU average (5 samples)	**0.6062**421	**0.6053**803
standard deviation	*0.0000397*	*0.0000228*
average error	*0.0000018*	*0.0000186*
CADNA (`float_st`)	0.6062	0.6053
error	*0.0000439*	*0.0000617*

Table 1: Roots of $7169x^2 - 8686x + 2631$, computed by different means.

k	u_k average
0	**2.000000**
1	**-4.000000**
2	**18.500000**
3	**9.378378**
4	**7.801148**
5	**7.154356**

k	u_k average
6	**6.805**962
7	**6.58**0517
8	**6.2**65057
9	*3.400501*
10	*-83.174968*
11	**1**14.316190

k	u_k average
12	**100**.777983
13	**100.0**47690
14	**100.00**2459
⋮	⋮
20	**100.000000**

Table 2: Initial iterates of sequence u_k computed by a Python program running under single precision MCA (5 samples) with VERROU.

4.1 Numerical benchmarks

We first assess the usability of VERROU on a few classical test cases, which are well enough understood for a complete analysis to be performed.

Following Kahan [18], we first consider trinomial $a\,x^2 - b\,x + c$ with $a = 7169$, $b = -8686$ and $c = 2631$, and define its roots

$$\{r_1, r_2\} = \frac{-b \pm \sqrt{b^2 - 4\,a\,c}}{2\,a}.$$

Exact solutions to this problem are reproduced in table 1, along with floating-point results obtained using a simple C program. The table shows results from a standard execution in IEEE-754 single precision (`float`) with rounding to nearest, as well as results obtained with the same program (*i.e.* the exact same executable binary) running under single-precision MCA using VERROU. In this latter case, statistics are collected from 5 samples, and bold digits are considered exact using estimation (1). For reference, results obtained with CADNA are also shown in the table. In this case, the program sources have been instrumented, and only exact significant digits (as evaluated by CADNA) are shown.

It is interesting to note that even for such a simple program, performing as few as 14 binary operations and 2 square roots calculations, a few MCA samples are enough to detect the occurrence of a problem. Obtaining a correct and reproducible estimation of inaccuracies would however have required far more sampling.

A second test case is the second-order recurrent sequence (u_k) defined by [19]

$$\begin{aligned} u_0 &= 2, \\ u_1 &= -4, \\ u_{k+1} &= 111 - \frac{1130}{u_k} + \frac{3000}{u_k\,u_{k-1}}, \quad \forall k > 0. \end{aligned}$$

Exact calculations show that $\lim_k u_k = 6$. However, when computing iterates of this sequence using IEEE-754 floating-point arithmetic with default rounding to nearest, we get very different results, converging to the value 100.

Table 2 presents results from such a computation, when run under single precision MCA using VERROU. The originality of this experience comes from the fact that the instrumented program was written in Python. No recompilation of any sort has had to take place: the VERROU command line was simply prepended to the call to the Python interpreter to produce the results shown.

Statistics are collected from 5 samples. Using formula (1), one can estimate that digits written in bold face are exact. Rows in slanted face in the table indicate intermediate iterates for which the estimated number of exact significant digits under MCA is less than 1. These results show that, when observing only a late iterate (such as u_{20}), MCA does not allow to detect any problem. It is interesting to note here that, although running a similar program under Discrete Stochastic Arithmetic (DSA) using CADNA would have yielded essentially the same results, the self-validation feature would at least have warned about the (in)validity of the results.

However, the problem appears very clearly when observing all iterates: the standard deviation starts increasing rapidly, before vanishing as the sequence converges to its attractive fixed point at 100.

	Reference Time	RANDOM 1	RANDOM 2	AVERAGE 1	AVERAGE 2
Case0	0.55s	13.89s (×25)	13.88s (×25)	15.04s (×27)	15.05s (×27)
ascan.dat		0.000000	0.000000	0.000000	0.000000
Case1	4.75s	91.44s (×19)	91.12s (×19)	96.75s (×20)	95.94s (×20)
ins1.dat		0.000006	0.000006	0.000006	0.000006
ascan.dat		0.000000	0.000000	0.000000	0.000000
Case2	0.45s	14.17s (×31)	14.20s (×31)	11.88s (×26)	11.79s (×26)
ins1.dat		0.280809	0.355187	0.105049	0.000000
enerloc.dat		0.234995	0.227002	0.086034	0.000000
sismo.dat		0.160344	0.196354	0.008062	0.000000
ascan.dat		0.290362	0.194622	0.344207	0.016042
Case3	11.33s	316.73s (×27)	315.42s (×27)	237.13s (×20)	237.52s (×20)
ascan.dat		0.000000	0.000000	0.000000	0.000000
ins2.dat		0.000000	0.000000	0.000000	0.000000
Case4	29.91s	1048.10s (×35)	1052.27s (×35)	1120.39s (×37)	1114.05s (×37)
sismo.dat		0.000000	0.000000	0.000000	0.000000
ascan.dat		0.000000	0.000000	0.000000	0.000000
Case5	10.48s	184.01s (×17)	183.46s (×17)	194.64s (×18)	193.14s (×18)
ins1.dat		0.000000	0.232258	0.000000	0.000000
enerloc.dat		0.000000	0.055204	0.000000	0.000000
ascan.dat		0.000000	0.244644	0.000000	0.000000
Case6	2.04s	24.97s (×12)	25.44s (×12)	26.67s (×13)	26.79s (×13)
Case7	0.80s	17.97s (×22)	18.11s (×22)	18.95s (×23)	18.76s (×23)
ins1.dat		0.000000	0.000000	0.000000	0.000000
ascan.dat		0.000000	0.000000	0.000000	0.000000
Case8	0.39s	147.77s (×379)	144.15s (×370)	60.65s (×155)	59.93s (×154)
Case9	10.31s	149.74s (×14)	147.56s (×14)	154.88s (×15)	155.58s (×15)
ascan.dat		0.000000	0.000000	0.000000	0.000000
Case10	21.28s	352.96s (×16)	347.29s (×16)	378.13s (×17)	374.82s (×17)
ins1.dat		0.000000	0.000005	0.000000	0.000005
sismo.dat		0.000000	0.000000	0.000000	0.000000
ascan.dat		0.000000	0.000000	0.000000	0.000000

Table 3: Output of the non-regression test suite of Athena2D for 2 runs with RANDOM mode and 2 with AVERAGE mode.

4.2 Verification & Validation of an industrial code

Ultra-sonic non-destructive testing is widely used to control steel welds. Athena [20] is a computing code designed to simulate the wave propagation inside the transducer with the Stepanishen equation [21] and in the steel with a finite element method [22]. After 15 years of development, Athena-2D (resp. Athena-3D) is now a complex software with 27 (resp. 55) thousand lines of code (95% of Fortran), depending on external libraries Blas, Lapack, MPI (3D only), and Umfpack (3D only). As the V&V process is crucial for Athena, a large number of comparisons to experimental results have been performed and non-regression testing tools have been put in place. However, due to legacy code and external libraries, it is far from easy to instrument sources using CADNA or to perform comparisons with higher precision results.

Using VERROU required only a few modifications to Athena: some small parts had to be uninstrumented using client requests in order to avoid breaking pre-conditions required by the algorithm (such as the $|\cos(x)| > 1$ or non-determinism problems mentioned in sections 3.5 and 3.7).

Following the scheme described in figure 3a, four executions (two with `RANDOM` and two with `AVERAGE` mode) of a short selection of the non-regression test suite of Athena-2D are reported in Table 3. Each output file contains a vector of results, of which the relative error is reported in 2-norm. Result files which are binary equal for all executions do not appear in the table. For example, **Case6** and **Case8** do not reveal any discrepancy. These two test cases output their results in ASCII format with 4 decimal places, which eliminates numerical noise less than 10^{-4}. While such treatments might be useful in some instances, they should be avoided when implementing accuracy assessment as described in figure 3b.

Case0, **Case1**, **Case3**, **Case4**, **Case7** and **Case10** reveal differences in the output files but the discrepancies are negligible (inferior to 10^{-5} relative difference). **Case2** and **Case5** reveal significant discrepancies which have to be analyzed. A close look of the execution traces and the output results shows that a threshold effect in the coupling between the transducer and the steel leads to a time shift (which can be easily visualized with existing tools: cf. figure 4). Since the amplitude of signals and the time between echos are not modified, users of Athena are not concerned by these discrepancies. However, developers are more disturbed as they have to manually check and understand such cases. To avoid this fastidious work, Athena's developers can improve the transducer–steel coupling to avoid threshold

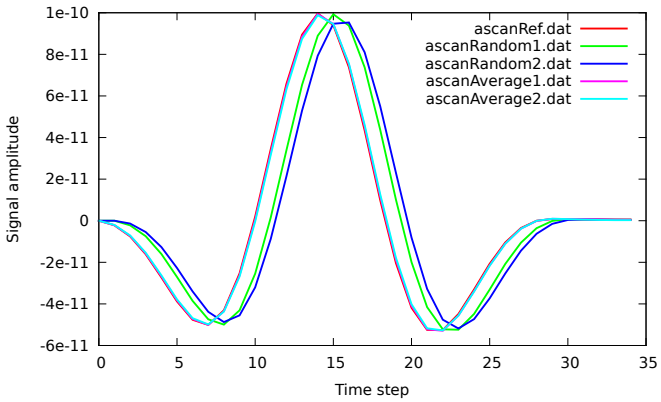


Figure 4: Comparaison of an ascan file for each execution of **Case2**.

effects, or develop norms adapted to user requirements. The first approach seems easier.

To be applicable in an industrial process, MCA also has to feature a reasonable run-time overhead. For Athena, compiled as used in production with `gfortran -O3`, the slow-down for one sample is usually between $\times 12$ and $\times 40$. The run-time overhead of **Case8** is surprisingly high in `AVERAGE` rounding mode, and even higher in `RANDOM` rounding mode. This case has two specificities. First there is an intensive use of trigonometric functions and, consequently, of client requests. Second, small matrices are assembled and stored only if the difference with neighboring matrices is not too small. As the criterion is conservative, around 40 times more matrices are stored and factorized in the `RANDOM` or `AVERAGE` rounding modes than in the default rounding to the nearest.

In Athena-3D, only 89% of floating point operations are instrumented due to the presence of single-precision SSE instructions. We therefore don't report numerical results in this paper. However, the experiment is still useful to demonstrate that external libraries such as MPI and Umfpack are taken into account without any difficulties, and that larger test cases (10^7 cells and 600 time steps) can be run with similar run-time overheads ($\times 25$).

The Athena experiment demonstrates that VERROU and non-regression tools can really easily be used together. Still, a strong knowledge of the industrial code is required to be able to analyze the results.

5 Conclusions

In this paper, we have shown that the MCA approach is adapted to diagnose inaccuracies due to floating-point arithmetic in numerical simulations. We proposed VERROU, an MCA implementation built upon the Valgrind framework, which does not require re-compiling the code nor its external dependencies. We showed that VERROU was suitable for the analysis of both small scale programs and large scale simulation codes. Thanks to a low entry cost, this new tool will be easier to promote in the industry as a preliminary diagnostic tool allowing detection of numerical errors at a coarse grain. This can then lead users to extract smaller parts of the application which reproduce the problem, and analyze them in more depth, either with VERROU itself, with tools previously cited in introduction such as CADNA, or with other arithmetics (Rational Number or Interval Arithmetic).

Acknowledgement

The authors feel greatly indebted to Josef Weidendorfer, who helpfully gave invaluable advice on valgrind tools development.

References

[1] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[2] J.-L. Lamotte, J.-M. Chesneaux, and F. Jézéquel, "CADNA_C: A version of CADNA for use with C or C++ programs," *Computer Physics Communications*, vol. 181, no. 11, pp. 1925–1926, 2010.

[3] S. Montan, "Sur la validation numérique des codes de calcul industriels," Ph.D. dissertation, Université Pierre et Marie Curie (Paris 6), France, 2013, in French.

[4] F. Benz, A. Hildebrandt, and S. Hack, "A dynamic program analysis to find floating-point accuracy problems," in *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[5] M. O. Lam, J. K. Hollingsworth, and G. Stewart, "Dynamic floating-point cancellation detection," *Parallel Computing*, vol. 39, no. 3, pp. 146–155, 2013.

[6] F. Févotte and B. Lathuilière, "VERROU: L'arithmétique stochastique sans recompiler," in *7ème Rencontre Arithmétique de l'Informatique Mathématique (RAIM)*, 2015, in French.

[7] D. Stott Parker, "Monte Carlo arithmetic: exploiting randomness in floating-point arithmetic," University of California, Los Angeles, Tech. Rep. CSD-970002, 1997.

[8] G. E. Forsythe, "Reprint of a note on rounding-off errors," *SIAM Review*, vol. 1, no. 1, pp. 66–67, 1959.

[9] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[10] M. Frechtling and P. H. W. Leong, "MCALIB: Measuring sensitivyty to rounding error with monte carlo programming," *ACM Transactions on Programming Languages and Systems*, vol. 37, no. 2, p. 5, 2015.

[11] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: a multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, 2007.

[12] W. V. Kahan, "The improbability of probabilistic error analyses for numerical computations," in *UC Berkeley Stastistics Colloquium*, 1996.

[13] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Reading, MA, USA: Addison-Wesley, 1997.

[14] T. J. Dekker, "A floating-point technique for extending the available precision," *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, 1971.

[15] A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why?" *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 253–267, Oct. 1999.

[16] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

[17] A. Zeller, *Why Programs Fail*, 2nd ed. Boston: Morgan Kaufmann, 2009.

[18] W. V. Kahan, "A survey of error analysis," in *Proc. IFIP Congress 1971*, 1971, pp. 200–206.

[19] J.-C. Bajard, D. Michelucci, J.-M. Moreau, and J.-M. Muller, "Introduction to the special issue "Real Numbers and Computers"," in *Journal of Universal Computer Science*. Springer, 1996, pp. 436–438.

[20] B. Chassignole, R. E. Guerjouma, M.-A. Ploix, and T. Fouquet, "Ultrasonic and structural characterization of anisotropic austenitic stainless steel welds: Towards a higher reliability in ultrasonic non-destructive testing," *{NDT} & E International*, vol. 43, no. 4, pp. 273 – 282, 2010.

[21] P. R. Stepanishen, "Transient radiation from pistons in an infinite planar baffle," *The Journal of the Acoustical Society of America*, vol. 49, no. 5B, pp. 1629–1638, 1971.

[22] E. Becache, P. Joly, and C. Tsogka, "Fictitious domains, mixed finite elements and perfectly matched layers for 2-d elastic wave propagation," *Journal of Computational Acoustics*, vol. 9, no. 3, pp. 1175–1201, 2001.