Gestion du compromis entre la performance et la précision de code de calcul

Nestor Demeure

université
PARIS-SACLAY

Compromise between precision and performance in high-performance computing.

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n° 574, mathématiques Hadamard (EDMH)
Spécialité de doctorat : Mathématiques appliquées
Unité de recherche : Université Paris-Saclay, CNRS, ENS Paris-Saclay, Centre Borelli, 91190, Gif-sur-Yvette, France.
Référent : ENS Paris-Saclay

Thèse présentée et soutenue en visioconférence totale, le 11 janvier 2021, par

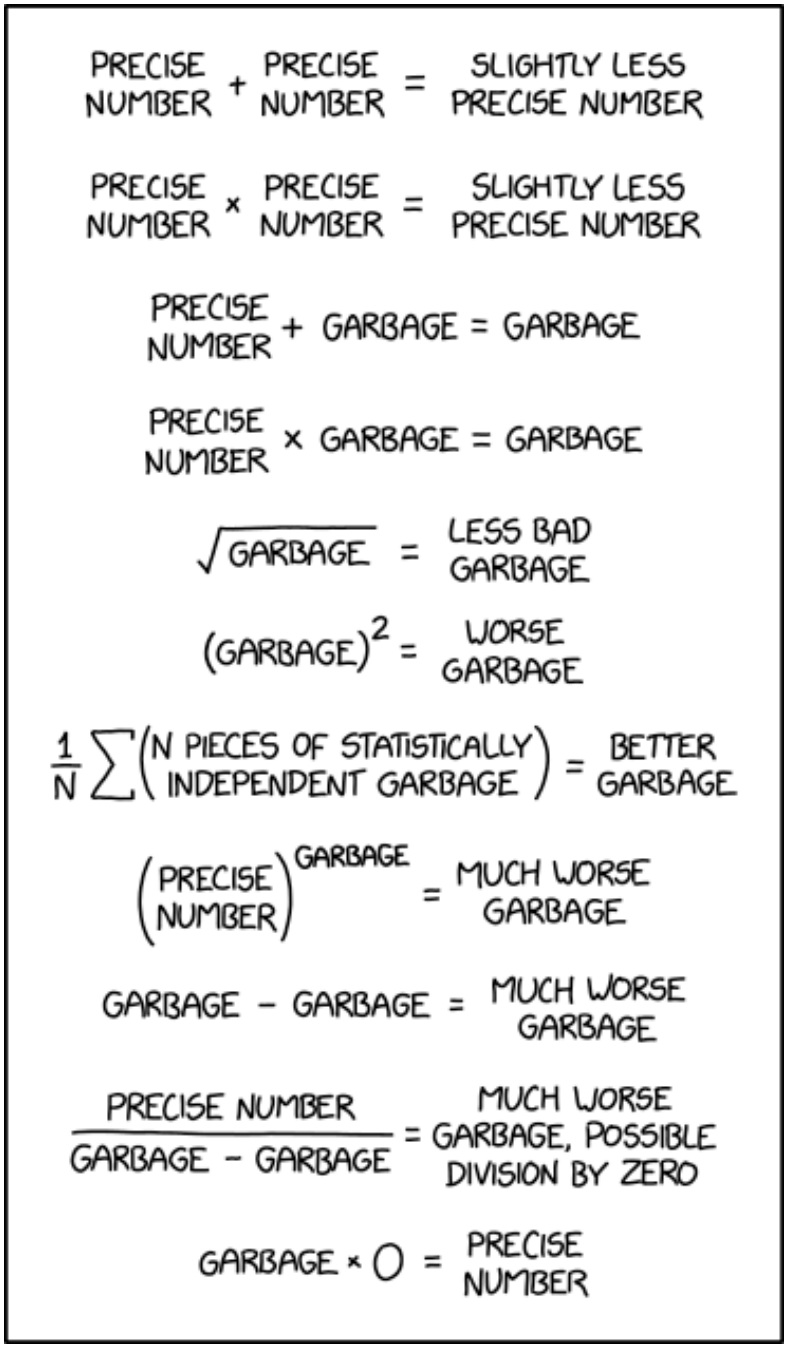
Nestor DEMEURE

Composition du jury :

Jean-Marie Chesneaux Inspecteur Général de l'Education, du Sport et de la Recherche	Président
Matthieu Martel Professeur, Université de Perpignan Via Domitia	Rapporteur
Jean-Michel Muller Directeur de recherches, École Normale Supérieure Lyon	Rapporteur
Jean-Yves L'Excellent Chargé de recherche, Mumps Technologies	Examinateur
Pablo de Oliveira Castro Maître de conférences, Université de Versailles-Saint-Quentin-en-Yvelines	Examinateur
Christophe Denis Maître de Conférences, Sorbonne Université	Directeur
Cédric Chevalier Ingénieur-Chercheur, CEA	Coencadrant
Pierre Dossantos-Uzarralde Chercheur, CEA	Coencadrant

Thèse de doctorat

NNT: 2021UPASM004



https://xkcd.com/2295/

Remerciements

À Pierre sans qui cette thèse n'aurait pas eu lieu.

À Christophe pour m'avoir fait découvrir le monde de l'erreur numérique.

À Cédric pour avoir toujours été là pour échanger sur ma thèse.

À Alexandre, Pierre, Vincent et Yitzhak pour leurs relectures attentives.

À tous les membres de mon jury pour leur rigueur et leurs questions stimulantes.

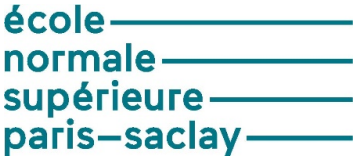
À tous les chercheurs de mon domaine avec qui j'ai eu le plaisir d'échanger.

À tous les doctorants de Ter@tec pour leur bonne humeur caractéristique, les croissants et les petits pains.

À l'équipe du centre Borelli pour m'avoir guidé à travers les labyrinthes administratifs.

À tous les employés du CEA pour leur soutient et leur curiosité envers mon sujet.

Et finalement à Elann pour m'avoir laissé emprunter quelques une de ses qualités pendant la rédaction de cette thèse.



Contents

Introduction

As Moore's laws is reaching its limit, the authors of computer programs cannot count on new processors to gain significant improvements in performance. Thus, they are now focusing on new avenues including different architectures, such as Graphical Processing Units (GPU), parallelism to get more power by adding more processors to the task, and mixed precision algorithms to improve the number of operations per second by reducing the working precision locally. All of those approaches — one might argue, the writing of high-performance numerical algorithms in general — requires a control over the trade-off between performance and precision. For example, using mixed precision requires the ability to consider which parts of a program could still produce useful outputs when computed in reduced precision. The link between performance and precision also appears when using parallel algorithms. These make the non-associativity of floating-point operations explicit by producing slightly different results from one run to another, which raises questions about the reproducibility of results. Finally, most GPUs can only reach their peak performance on 32 bit precision — or even 16 bit, half, precision — while the majority of current scientific software is developed in 64 bit floating-point arithmetic. Thus, using those approaches requires knowing how accurate our results are before and after their application. The aim of this thesis is to give tools to users so they can make informed decisions about the precision they are trading for performance, with a focus on simulation and high-performance computing. While I concentrated my efforts on numerical accuracy, I will also detail an approach for the selection of a linear solver and preconditioner in Part IV.

My original research question was determining whether it is possible to increase the performance of a program by decreasing precision locally without compromising the accuracy of its outputs, but I quickly discovered a lack of viable methods to measure the numerical accuracy on my problems of interest. Further research revealed that the systematic verification of the numerical accuracy of an application is not a common practice, meaning that, often, only the most glaring numerical problems are likely to be caught. I thus decided to focus on the measure of numerical error and, in particular, its application to simulation and high performance computing: large programs that can have long run times and a high number of operations per

second, thus requiring a method that scales well. High performance computing is of particular interest because, as computing power has increased, so did the size of simulations, their resolution, the number of their arithmetic operations, and the magnitude of the values they process. We can now run more than 10^{15} floating-point operations per second on a supercomputer and 10^{12} floating-point operations per second on a standard GPU. In this context, one can expect the rounding errors introduced by the use of floating-point arithmetic to have an increasing impact on simulations. This matters as numerical errors can significantly degrade numerical results, but also introduce artifacts in physical simulations and cause phenomena to be missed or misinterpreted [Bailey and Borwein, 2015].

As detailed in Chapter 3, evaluating the numerical accuracy of a program is a hard problem and, while good solutions have been developed, they are either not suited for very large programs due to their working hypotheses or very slow and difficult to interpret results. Solutions to the next logical problem, identifying the sources of numerical error in an inaccurate program, suffer from the same shortcomings. My core contribution is the development and validation of a new method, which I call *encapsulated error*, to measure the numerical error of a program and localize its sources of error. A one line summary of the method would be: we evaluate the numerical error produced locally and track its propagation in further computations via a dedicated type that encapsulate both the result of the original computation and an approximation of its numerical error. This gives us direct access to the number of significant digits at all times and for all intermediate results. While simple, this method is competitive with the state of the art, compatible with various forms of parallelism, and has a low overhead. This makes it suitable for use with large programs as illustrated on various applications from the field of physics simulation. Figure 1 illustrates an application of our method to a simple test case that will be detailed in Section 6.2.2. Furthermore, it requires building blocks that are readily available in most programming languages, making it easy to implement (we provide reference implementations, under the name *Shaman* and *Shaman_julia*, in both the C++ and Julia programming languages), and gives easily interpretable results that can be analyzed by a non-expert, an important property as most numerical codes are not written by numerical analysts. It is my hope that this method will make the analysis of the numerical accuracy of a program both more accessible and viable for a wider range of applications.

This dissertation is split into five parts. The first part gives a proper definition of numerical error, details our hypotheses (and in particular the use of the IEEE-754 standard), and covers the state of the art of measuring numerical error and localizing the sources of error in a program. The second part covers our main contribution: our method to measure and locate the sources of numerical error, how it works, how it is implemented, and why it works. The third part checks both the accuracy

of the method and its runtime overhead, compared to the state of the art, on a variety of test cases of varying complexity. The fourth part details the numerical analysis of three large programs coming from the field of physics simulation. It also illustrates the interactions between numerical error and uncertainty. The final part is particular in that it does not deal with numerical error, but instead goes back to the trade-off between precision and performance. It explores an interesting use of machine learning, predicting the convergence profile of a linear solver for a new linear system, which can be used to select faster, more accurate solvers and preconditioners in order to improve results, while still using a properly proven solver.

I made the conscious decision of writing most of this text in English, except for an extended summary (page 153). It is not my mother tongue, but, paradoxically is, the *lingua franca* of contemporary research. I have enjoyed reading the manuscripts of Scandinavian, German, and Chinese authors in English, one of the few languages I can read. It is time to return the favor.

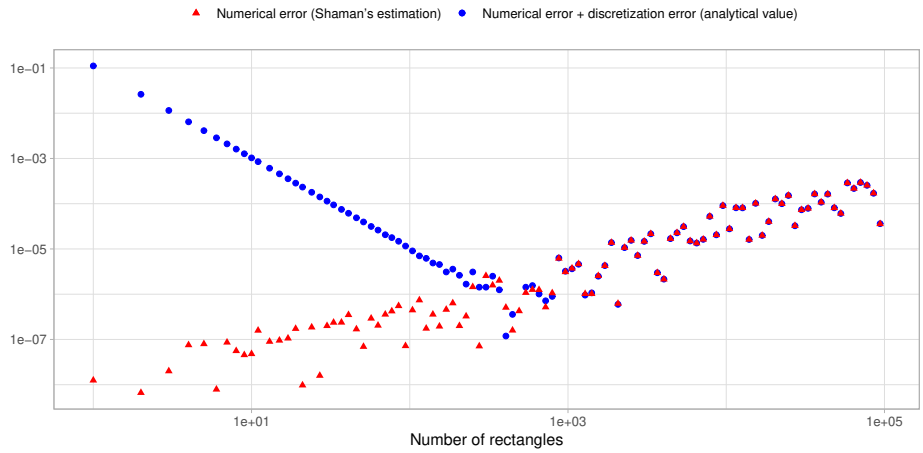


Figure 1: Integrating the cosine function between 0 and $\frac{\pi}{2}$ using the rectangle rule. Both axes are displayed on a logarithmic scale. The analytically computed total error, discretization error plus numerical error, is plotted in blue, while our estimation of the numerical error is plotted in red. As the number of rectangles increase, the discretization error converges toward zero, leaving us with the numerical error due to the sum of areas.

Part I

Numerical Error

Table of Contents

Chapter 1

Floating-point numbers

To manipulate real numbers with a computer, we first need to encode them. This is usually done by a surjective mapping between $\mathbb{R}$, the set of real numbers, and a subset $\mathbb{F}$ of representable numbers. It is the mismatch between those sets that is at the root of the numerical error studied in this dissertation.

This chapter covers common computer representations for real numbers before focusing on floating-point arithmetic and, in particular, the IEEE-754 standard [IEEE, 2019] which is the current standard in simulation and high-performance computing.

1.1 Computer representation of reals

There are infinitely many reals, but with a fixed number of bits, n, you can represent at most 2^n different numbers. This implies that using a computer to represent numbers requires a trade-off between the range of numbers that can be represented accurately, memory use, and performance.

In this section we introduce common computer representations for real numbers and their usual use-cases. Note that the representation of numbers is still an active area of research, recent evolutions include the democratization of half-precision, 16 bit floating-point arithmetic, and the introduction of posit numbers [Gustafson, 2015].

1.1.1 Exact representations

It is possible to encode some subsets of real numbers, such as rationals and algebraic numbers, exactly. Methods to do so include arbitrary-precision numbers (allocating memory for more bits when needed), fractions (building on large integers and restricting the field of available operations), and symbolic computations (which

might require storing every intermediate operation to determine the n^{th} digit of the final output after the fact).

The most common exact number representation is probably rational numbers. Rational numbers, encoded as a pair of arbitrary-precision integers (also called *bignum integers*), form a field: the addition, subtraction, multiplication and division of rationals are all exact since their output can be encoded as a ratio of integers. Note that we are only encoding a subset of reals (which, for example, does not include π) and that we cannot apply arbitrary mathematical operations to this representation as, for example, the cosine of a non-zero rational is never a rational.

Exact representations usually require a variable memory consumption and are much less CPU efficient [Strzeboński, 1997, Boehm, 2020]. However, those representations are used in fields such as mathematics (where one might want to evaluate the terms of a series exactly) and computer geometry (to guarantee the absence of artifacts and check mathematical properties).

1.1.2 Fixed-point representations

A very straightforward way to encode real numbers is to use a fixed-point representation. A number x is then represented by a sign s and an integer f such that $x = s \times f \times b^{-q}$ where b is the basis (usually binary or decimal) and q is a fixed, predetermined, exponent (hence the name fixed-point). This representation makes fixed-point numbers easy to emulate with integers which is useful when one cannot rely on a hardware implementation.

This representation fixes the absolute error: numbers are regularly spaced along the real line such that the maximum absolute error introduced when rounding a number is a constant, it will be identical whether we are manipulation 0 or 10^{10}.

A strong downside of this representation is its limited range making it unpractical to represent common physical constant such as speed of light (≈ 300000000 m/s) and Newton's gravitational constant ($\approx 0.0000000000667\ \mathrm{N} \cdot \mathrm{m}^2 \cdot \mathrm{kg}^{-2}$) with the same encoding and meaning that multiplications and divisions tend to quickly result in overflow or underflow.

Fixed-point numbers appear mostly in finance where the numeric range are limited and regulations are often expressed in terms of absolute error, embedded systems where processors usually lack floating-point instruction and signal processing to reduce costs by using less expensive processors.

1.1.3 Floating-point representations

Floating-point numbers, whose representation is detailed in section 1.2.2, encode reals with two numbers, one of them being the exponent, which is implicit and fixed with fixed-point numbers.

This means that representable numbers cover a very wide range of powers, the double precision format was designed to be able to express basic physical constants, but the spacing between two representable numbers is now variable. Numbers around zero are now much closer to one another than numbers around very large values.

Floating-point numbers are, currently, the most common representation used in numeric applications. In the following we will focus on them, and in particular on the IEEE-754 formats [IEEE, 2019].

1.2 The IEEE-754 standard

One can make many design decisions while designing a floating-point number format. This section gives a brief overview of the various floating-point formats available before the introduction of the IEEE-754 standard, motivating its introduction, followed by a description of the parts of the specification that are of interest for our work (we recommend [Muller et al., 2010] for a thorough discussion of IEEE-754 floating-point arithmetic and its properties). It finishes with a description of the consequences of the existence of a standard for numerical analysis.

1.2.1 Genesis of the standard

Before 1985, as can be seen in Table 1.1, floating-point formats were mostly manufacturer-dependent and varied widely. This made results non-reproducible between architectures and the numerical stability of algorithms hard to predict [Kahan, 1997, Severance, 1998].

This problem was solved by the introduction of the IEEE-754 standard [IEEE, 2019]. The first version of the standard was developed by a committee headed by William Kahan [Kahan, 1997]. Published in 1985 and revised in 2008 and 2019, it specifies various formats for the binary representation of reals that are now standard and implemented by the vast majority of processors[1].

This was an effort to both standardize floating-point representations and build on a more principled basis to ensure properties such as the preservation of the commutativity of the operations (however associativity and distributivity are lost as detailed in section 2.2.1) and the fact that $x - y = 0 \iff x = y$ which was not guaranteed by most previous implementations due to small values being flushed to zero in the absence of subnormal numbers [Severance, 1998].

[1] Graphics processing unit (GPU) are the most common exception. They usually implement only a subset of the standard, some of them having no denormals or only some rounding modes [Whitehead and Fit-Florea, 2011].

Computer	Base (b)	Significand's digits in the base (p)	Machine epsilon ($\frac{1}{2}b^{1-p}$)	Emin	Emax
Univac 1108	2	27	$2^{-27} \approx 7.45e^{-9}$	-128	127
Honeywell 6000	2	27	$2^{-27} \approx 7.45e^{-9}$	-128	127
PDP-11	2	24	$2^{-24} \approx 5.96e^{-8}$	-128	127
Control Data 6600	2	48	$2^{-48} \approx 3.55e^{-15}$	-975	1070
Cray-1	2	48	$2^{-48} \approx 3.55e^{-15}$	-16384	8191
Illiac-IV	2	48	$2^{-48} \approx 3.55e^{-15}$	-16384	16383
SETUN	3	18	$\frac{3^{-17}}{2} \approx 3.87e^{-9}$	?	?
Burroughs B5500	8	13	$2^{-37} \approx 7.28e^{-12}$	-51	77
Hewlett Packard HP-45	10	10	$5e^{-10}$	-98	100
Texas Instruments SR-5x	10	12	$5e^{-12}$	-98	100
IBM 360 and 370	16	6	$2^{-21} \approx 4.77e^{-7}$	-64	63
IBM 360 and 370	16	14	$2^{-53} \approx 1.11e^{-16}$	-64	63
Telefunken TR440	16	$9\frac{1}{2}$	$2^{-35} \approx 2.91e^{-11}$	-127	127
Maniac II	65536	$2\frac{11}{16}$	$2^{-17\frac{11}{16}} \approx 3.73e^{-9}$	-7	7

Table 1.1: Floating-point format of various machines developed before the publication of the IEEE-754 standard. Extracted from [Forsythe et al., 1977] (see section 1.2.2 for an explanation of the column names).

1.2.2 Formats

The IEEE-754 standard defines a variety of floating-point formats and associated sets of representable numbers, listed in Table 1.2, as (b, p, $[Emin, Emax]$) triplets where b is the base in which numbers will be encoded, p is the precision, the number of digits of the *significand* and, $[Emin, Emax]$ is an exponent range which defines the range of values covered by the *exponent*.

For a given format, representable numbers are encoded as triplets (s, q, f) composed of:

- A *sign bit* $s \in \{0, 1\}$.
- An *exponent* q encoded as an unsigned integer such that $Emin \leq q - Emax \leq Emax$. To decode it, one has to subtract the bias $Emax$ from q which enables the representation of negative powers.
- A *significand*, also called *fraction* or *mantissa*, f encoded as a p digit unsigned

integer in base b. To decode it, one needs to interpret it as a fraction in the given base with an implicit one in the first place (unless the number is zero or a subnormal). Thus the fraction 011 would be decoded as the significand 1.011 which, in binary, is $2^0 + 2^{-2} + 2^{-3} = 1.375$.

Name	Common name	Base (b)	Significand's digits in the base (p)	Machine epsilon ($\frac{1}{2}b^{1-p}$)	Emin	Emax
binary16	Half precision	2	11	$2^{-11} \approx 4.88e^{-4}$	-14	+15
binary32	Single precision	2	24	$2^{-24} \approx 5.96e^{-8}$	-126	+127
binary64	Double precision	2	53	$2^{-53} \approx 1.11e^{-16}$	-1022	+1023
binary128	Quadruple precision	2	113	$2^{-64} \approx 5.42e^{-20}$	-16382	+16383
binary256	Octuple precision	2	237	$2^{-113} \approx 9.63e^{-35}$	-262142	+262143
decimal32		10	7	$5e^{-7}$	-95	+96
decimal64		10	16	$5e^{-16}$	-383	+384
decimal128		10	34	$5e^{-34}$	-6143	+6144

Table 1.2: IEEE-754 floating-point formats.

Numbers can be decoded using the formula $x = (-1)^s \times f \times b^q$. The single precision number represented in Figure 1.1 is decoded as follow:

$$
\begin{aligned}
&s = 0 \\
&f = 2^0 + 2^{-2} = 1.25 \\
&q = (2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7) - 127 = -3 \\
&x = (-1)^s \times f \times b^q = (-1)^0 \times 1.25 \times 2^{-3} = 0.15625
\end{aligned}
$$

A handful of special cases are added to this representation in order to represent exceptions and specific numbers:

- $+0$ and -0, which are represented by setting all the bits to zero except for the sign bit.

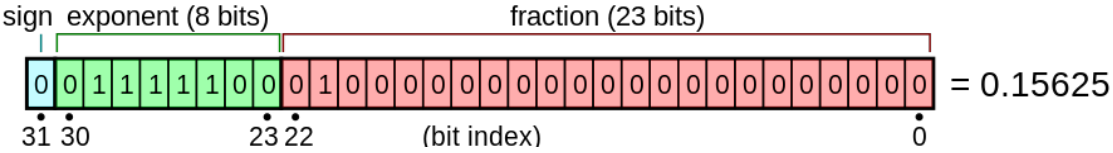


Figure 1.1: Binary representation of 0.15625 in IEEE-754 32-bit floating-point arithmetic. Picture extracted from [Stannered, 2008].

- $+\infty$, $-\infty$, which are represented by setting all the exponent bits to one and all the fraction bits to zero and can be produced by operations such as the division of a finite, non-zero, number by zero or an overflow.
- two kinds of *Not-a-number* (*NaN*), signaling and silent (also called quiet), to encode the output of invalid operations such as taking the square root of a negative number or dividing zero by zero. They are represented by setting all the exponent bits to one and setting some[2] bits of the fraction to one.
- *subnormal* (also called *denormal*) numbers, used to have a finer granularity around zero and avoid flushing results to zero. They are encoded by setting the exponent bit set to zero which implies that the implicit first bit of the fraction should be set to zero.

1.2.3 Rounding modes

Given a real $x \in \mathbb{R}$, which can be either an input or the result of an operation done on one or more representable numbers, one needs a surjective mapping $\mathbb{R} \to \mathbb{F}$. To do so, the IEEE-754 standard specifies six possible rounding modes which select the next or previous representable number as a function of x:

- *Round to nearest, ties to even.* Rounding to the nearest representable number, break ties by selecting the representable number with an even last bit to prevent bias toward larger or smaller numbers. We will denoted it $\uparrow_n (x)$.
- *Round to nearest, ties away from zero.* Rounding to the nearest representable number, break ties by selecting the representable number with the larger magnitude. We will denote it $\uparrow_{n_a} (x)$.
- *Round to nearest, ties to zero.* Rounding to the nearest representable number, break ties by selecting the representable number closest to zero. We will denote it $\uparrow_{n_0} (x)$.

[2]On x86 processors, silent NaN are denoted by setting the most significant bit of the fraction to one. However, this is not a general rule as the encoding of signaling and silent Nan is not specified by the IEEE-754 standard.

- *Round toward 0.* Rounding to the representable number closest to zero. We will denote it $\uparrow_0 (x)$.
- *Round toward* $+\infty$. Rounding to the representable number closest to $+\infty$. We will denote it $\uparrow_{+\infty} (x)$.
- *Round toward* $-\infty$. Rounding to the representable number closest to $-\infty$. We will denote it $\uparrow_{-\infty} (x)$.

The standard impose that processors default to *Round to nearest, ties to even* unless specified by the program. This rounding mode will be shortened as *Round to nearest* in the following discussion. The alternative rounding modes being mostly used in interval arithmetic implementations and within some specialized mathematical libraries.

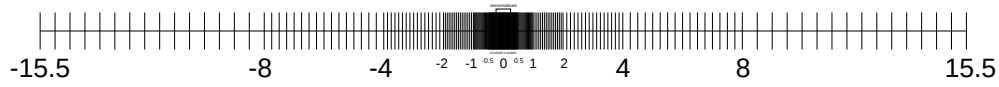


Figure 1.2: Distribution of representable floating-point numbers (black bars on the figure) for a hypothetical 8 bit floating-point format that has a 4 bit mantissa and a 3 bit exponent. The unit in the last place associated to a number is the distance between the two closest bars. Extracted from [Schatz, 2014].

One important quantity is the *unit in the last place* of a number (abbreviated *ULP*). Borrowing a definition from [Muller, 2005], if x is a real number that lies between two finite consecutive representable numbers $(x_1, x_2) \in \mathbb{F}^2$, without being equal to one of them, then $ULP(x) = |x_2 - x_1|$. Otherwise, $ULP(x)$ is the distance between the two finite consecutive representable numbers nearest x (*ULP(NaN)* is *NaN*).

This quantity, which can be seen in Figure 1.2, grows with the amplitude of x (larger representable numbers are further apart) and is used as a unit of accuracy as an operation cannot be more accurate than $\frac{1}{2}$ units in the last place since there would be no representable number to encode the result. All rounding modes defined by the standard guarantee that the rounding error will be lower than or equal to one ULP and given the round-to-nearest rounding mode, a real $X \in \mathbb{R}$ and a finite floating-point number $x \in \mathbb{F}$, we have:

$$\uparrow_n (X) = x \implies |X - x| \leq \frac{1}{2} ULP(x)$$

All arithmetic operators ($+$, $-$, $\times$, $/$) round according to the current rounding mode. While not mandatory, correct rounding is also recommended for some

common functions including the exponential, logarithm, sine, cosine, and tangent. This means that they can be treated as atomic operations, like arithmetic operators, that produce a result within one ULP of the output of an equivalent infinite precision operation applied to the same outputs. This property is a cornerstone of floating-point analysis as it gives an upper bound on the absolute difference between the output of a computation done with floating-points and the output of the same computation done with real numbers. Given three finite floating-point numbers $(x, y, z) \in \mathbb{F}^3$ and a real numbers $Z \in \mathbb{R}$, we have:

$$\left.\begin{array}{r} x + y = Z \\ \uparrow_n (Z) = z \end{array}\right\} \implies |Z - z| \leq \frac{1}{2} ULP(z) \implies |(x + y) - z| \leq \frac{1}{2} ULP(z)$$

For non-subnormal numbers, an ULP can be translated into a relative bound using $\epsilon = \frac{1}{2} b^{1-p}$, the *machine epsilon*, the smallest floating-point number that can be added to 1 in order to get a result different from 1 (subnormal numbers require a constant, not relative, bound):

$$\epsilon x \leq ULP(x) \leq 2\epsilon x$$

1.2.4 Consequences

The IEEE-754 standard and its wide application has had a profound impact on the reproducibility of numerical computations across hardware. Given a working precision, rounding mode and sequence of operations, all fixed at the assembly level, the standard guarantees that a computation will produce identical results from one CPU to another making it possible to develop numerical algorithms that are reliable across languages and processors. Furthermore, guarantees on the upper bound of the error introduced by the rounding of the arithmetic operators mean that computer scientists are able to reason and write proofs about floating-point computations, opening the door for error-free transformations (see section 2.4) and a wide range of static analysis (mentioned in section 3.1.2).

However, things are not as straightforward as it may seem. Most programming languages do not give access to all of the features provided by the standard [Kahan, 1997], such as the possibility to change the rounding mode of the processor. Some programming languages, such as Java [Kahan and Darcy, 1998], explicitly do not try to follow the standard. Even when one is using a programming language which follows the standard and gives ways to access most of its features, such as C or C++, many factors like parallelism and compiler optimization, which can change the working precision and reorder operations, can invalidate its properties and break reproducibility from one processor to the other. For example, the Intel C++

Compiler is notorious for reordering arithmetic operations by default, despite the non-associativy of floating-point arithmetic, which breaks some algorithms such as Kahan summation [Intel, 2013].

For the sake of simplicity, numbers encoded following the IEEE-754 standard will be referred to as *floating-point numbers* and *floating-point arithmetic* will be used as a synonym for the IEEE-754 floating-point arithmetic used with the, default, *round to nearest, ties to even* rounding mode.

Chapter 2

Definition and properties of numerical error

IEEE-754 floating-point arithmetic uses a finite set of numbers to represent reals. Hence, most numbers have no exact representations and the result of some arithmetic operations between floating-point numbers cannot be represented exactly as a floating-point. This round-off error, which we call numerical error, can accumulate and be propagated across computations, impacting the final result.

In this chapter, we define the various kinds of error that can affect a simulation before focusing on numerical error, which is illustrated with various examples, and introduce two related concepts: *Error-free transformations* and *significant digits*.

2.1 Types of error

There are four main types of error that can make the output of a simulation deviate from the physical truth:

- *Modeling error*, when the model is inaccurate or incomplete.
- *Discretization error*, when the algorithm has a step size that is too coarse for the targeted phenomena. We include temporal and spatial step sizes in this category but also the error due to the stopping criteria of iterative algorithms. This error goes to zero as the step size decreases (as seen in Figure 1).
- *Uncertainty* on the inputs and constants. This stochastic quantity models imperfect knowledge of the parameters of the model.
- *Numerical error*, the error introduced by the use of a non-exact computer representation for the numbers.

An example encompassing all those types of error would be the simulation of the solar system using the gravity between the planets and the sun as the only forces (Equation 2.1), Newton's second law to convert those forces into accelerations (Equation 2.2) and finite differences with a step ϵ to convert acceleration into speed into position (Equation 2.2, 2.3 and 2.4).

In this example, modeling error comes from using an imperfect formula for the gravity (Equation 2.1 does not take relativistic effects into account) and missing some forces (such as the gravity due to stars out of the solar system). Our discretization error comes from the parameter ϵ used for the finite differences (Equation 2.2, 2.3 and 2.4) and should reach zero as ϵ decreases in magnitude. Our uncertainty is the imprecision in our knowledge of the various constants such as the initial positions $\vec{x_0}$, speeds $\vec{v_t}$ and masses m_i of the planets. The numerical error is the error introduced by the use of a limited precision computer representation for numbers which introduce round-offs in the result of the various arithmetic operations such as the sum of forces in Equation 2.2.

$$\vec{f_{12}} = \frac{G_{m_1 m_2}}{d_{12}^2} \vec{u_{21}} \tag{2.1}$$

$$\vec{a_{t+\epsilon}} = \frac{\sum \vec{f}}{m} = \frac{\vec{v_{t+\epsilon}} - \vec{v_t}}{\epsilon} \tag{2.2}$$

$$\vec{v_{t+\epsilon}} = \vec{v_t} + \epsilon \vec{a_{t+\epsilon}} = \frac{\vec{x_{t+\epsilon}} - \vec{x_t}}{\epsilon} \tag{2.3}$$

$$\vec{x_{t+\epsilon}} = \vec{x_t} + \epsilon \vec{v_{t+\epsilon}} \tag{2.4}$$

Most users, and indeed most papers, assume that numerical error is a negligible quantity when compared to other sources of error [Yeo, 2020, Eça et al., 2019, Zikanov, 2019, Slater, 2008][1]. Some also believe that it should always be about 10^{-15} when using double precision[2]. These misconceptions seem to stem from the fact that the relative round-off error introduced by a single arithmetic operation in double precision should indeed be bounded by the machine epsilon which is 2^{-53} (about 1.11×10^{-16}). Furthermore, current algorithms tend to be built on decades of numerical analysis and to have good numerical properties, keeping the numerical

[1] As Oleg Zikanov said in [Zikanov, 2019, section 13.2.1]: *"The commonly accepted criterion is that the iteration errors are at least one order of magnitude lower than the discretization errors."*. Here the concept of *iteration error* includes numerical error and all other sources of error that differentiate the computed result from the theoretical output of the algorithm. It is to be noted that the author is, himself, careful with the notion and dedicates a section of his book to the numerical error dampening or amplifying properties of numerical schemes.

[2] The following citation from the introduction of [Eça et al., 2019] is particularly representative of a state of mind we found in many papers: *"We are assuming that double-precision (14 digits) is sufficient to obtain a negligible contribution of the round-off error to the numerical error."*.

error in check. However, as will be seen in chapter 8, we found that numerical error can still have a significant impact on real programs.

2.2 Numerical error

2.2.1 A definition of numerical error

Numerical error is introduced when a number is rounded to the closest computer-representable number. An example of rounding is the number 0.1 which, similarly to $\frac{1}{3}$ in decimal, would requires an infinite number of bits to be represented exactly in base two (0.00011001100110011...). Having only a limited number of bits available, one has to truncate the representation, thus 0.1 is encoded as 0.0001100110011 in half-precision which is equal to 0.0999755859375 in decimal. This is called a *round-off*, it can happen when encoding a real into a computer-representable format, when translating between two formats. Round-off can also occur when the output of an operation which takes representable numbers as inputs cannot be represented in the current format.

Round-offs are propagated through the computation leading to visible differences between computations done with real numbers and floating-points as seen in code listing 2.1.

```
(0.1 + 0.2) + 0.3 = (0.3 + error1) + 0.3
(0.1 + 0.2) + 0.3 = 0.6 + error1 + error2
(0.1 + 0.2) + 0.3 = 0.60000000000000009

0.1 + (0.2 + 0.3) = 0.59999999999999998
```

Code listing 2.1: Floating-point computation, done with two different parenthesizing, in double precision.

We call those differences *numerical errors*. Some publications include uncertainties in their definition of numerical error, to avoid ambiguities they use the term *round-off error*[3] to name the error produced by floating-point arithmetic. In the following numerical error and round-off error will be synonymous.

It is essential to discriminate between three slightly different definitions of numerical error. Given an expected result $X \in \mathbb{R}$ and the computed result $x \in \mathbb{F}$, they are defined as follows:

- The *numerical error* which we define as the difference between the result computed in infinite precision and the actual floating-point result, $E(X, x) =$

[3]The term *truncation error* is also used in this context. However, it is most commonly used to describe the error introduced by the truncation of an infinite sum and the discretization error.

$X - x$. This quantity is *signed* and entirely determined by a sequence of operations on given inputs and not stochastic, unlike uncertainty. It is the quantity that we measure and manipulate in the following.

- The *absolute numerical error* which is the absolute value of the numerical error, $E_{absolute}(X, x) = |X - x|$. It is unsigned, and thus its manipulation over several operations can only lead to an upper bound.

- The *relative numerical error* which is the absolute value of the numerical error divided by the computed result, $E_{relative}(X, x) = |\frac{X-x}{x}|$. IEEE-754 floating-point arithmetic operators try to minimize this quantity locally while, by contrast, fixed-point arithmetic will try to minimize the absolute error.

2.2.2 Stability, accuracy and precision

Various terms, most notably *stable*, *accurate* and *precise*, are employed to describe the numerical behavior of an algorithm or the quality of a result. In the context of floating-point analysis, the *stability* of an algorithm describes the way it is influenced by the use of finite precision, whether the numerical error in the intermediate steps is dampened or, on the contrary, magnified.

Following the notations of [Chesneaux et al., 2009], let's take an algorithm G, computed in $\mathbb{R}$, such that $Y = G(X)$ with $(X, Y) \in \mathbb{R}^2$ and the corresponding algorithm g, computed in $\mathbb{F}$, such that $y = g(X)$ with $y \in \mathbb{F}$. The *forward error* corresponds to the numerical error, namely $Y - y$, while the *backward error* is the smallest perturbation δ_X such that $y = g(X + \delta_X)$ (this concept, made famous by [Wilkinson, 1964], relates to the *residual* which is often the error measure used in linear algebra due to its ease of computation in this context).

A solution is said to be *accurate* if it has a low forward error, an algorithm is said to be *forward stable* if its *forward error* is of order $o(|y| \times \epsilon)$ and *backward stable* if its *backward error* is of order $o(|x| \times \epsilon)$. In the following, we will use the term *numerically stable* and, by opposition, *numerically unstable* to speak of *forward stable algorithms* and *precise* (compared to a result that would have been computed in $\mathbb{R}$) as a synonym for *accurate*.

2.2.3 Significant digits

One metric used to evaluate the quality of a numerical result is its *number of significant digits*. However, as discussed in [Parker, 1997, section 4.1] and [Sterbenz, 1974, section 3.1], the concept of significant digits is usually not formally defined, due to its perceived intuitive nature. This leads to non-obvious results. An example (taken from [Parker, 1997]) would be that if one expects a computation to produce

3.14159, most people would consider 3.1415 to have five significant digits and 3.1416 to have four significant digits despite the fact that the later is closer to the expected result.

Following [Parker, 1997], we define the number of significant digits of a result as a function of an associated error and in base b, as:

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

This definition gives sensible, comparable, results for various kinds of error such as a signed numerical error, but also the width of an interval or the standard deviation of a distribution. The formula also has the interesting property of being very resilient to imprecision in the measure of the error: as we round the logarithm of a ratio, having the correct order of magnitude for the error is enough to get the correct result.

2.3 Some consequences of numerical error

2.3.1 Direct consequences

A direct consequence of numerical error is that, since the order of round-offs change with the order of operations, addition and multiplication are *not associative* in floating-point arithmetic as seen above in example 2.1. This is often detected in the form of varying outputs when running parallel programs as they tend to change the order of operations from one run to the other. It can also be seen when two mathematically equivalent formulae give different results when applied to floating-point numbers. One such example is the computation of the variance, Equation 2.5 and Equation 2.6 are mathematically equivalent, but Equation 2.5 (which is commonly used as it can be computed in a single pass on the data) can sometimes produce a negative variance (which should be impossible in $\mathbb{R}$) due to the accumulation of numerical errors surfacing in the final subtraction.

$$variance(x) = \left(\frac{1}{n}\sum_{i=1}^{n} x_i^2\right) - \left(\frac{1}{n}\sum_{i=1}^{n} x_i\right)^2 \tag{2.5}$$

$$variance(x) = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \frac{1}{n}\sum_{i=1}^{n} x_i\right)^2 \tag{2.6}$$

This particular case of round-off, a subtraction between two numbers of similar magnitude, where a single operation can have a drastic impact is called a *cancellation*. Cancellations lead to a drop in magnitude and thus the loss of several bits of precision. Another example of cancellation is the subtraction in $(10^{65} + 1) - 10^{65}$ which is evaluated as 0 in 64 bit IEEE-754 arithmetic since $10^{65} + 1$ rounds to 10^{65}. The subtraction is technically exact (as $10^{65} - 10^{65}$ is indeed 0) but, due to the drop in magnitude, the error from the initial round-off becomes apparent. Cancellations are sometimes called *catastrophic cancellation* when the number of bits lost becomes large enough (different tool will have different thresholds to define the point where a cancellation is considered catastrophic).

Another commonly observed consequence of numerical error is the presence of *unstable tests*, Boolean operators $(=, \neq, \leq, <, \geq, >)$ that would have produced different results if the full computation had been done with real numbers. The simplest, and most commonly known example, is the fact that unless proven otherwise one should avoid equality tests between outputs of floating point computations (as seen in example 2.1 where we observe that $(0.1 + 0.2) + 0.3 \neq 0.1 + (0.2 + 0.3)$). The most common solution is to, instead, test whether the absolute difference between the numbers is below a, problem dependent, acceptability threshold. In the following, we consider a test unstable if the difference of its operands has no significant digits when taking into account their respective numerical error (we define significant digits in section 2.2.3). As will be seen in section 5.3, even a comparison such as the convergence criteria of an iterative algorithm or, worse, a criterion used to decide on the model that will be used for a simulation, can be unstable leading to potentially significant differences between the output of a computation done with reals or floating-point numbers.

2.3.2 Examples of inaccurate results

This section presents some examples of inaccurate results to give the reader a feel for the kind of impact that numerical error can have on a computation.

W. Kahan's second order equation

The example given in code listing 2.2 was introduced by William Kahan [Kahan, 2004] and solves Equation 2.7 using the quadratic formula.

$$94906265.625x^2 - 189812534x + 94906268.375 = 0 \tag{2.7}$$

Using the resolution of a second degree polynomial, it illustrates how a well known, textbook formula (which most people are likely to use) can be numerically

unreliable.

```
double SecondOrderEquation()
{
    // a*x^2 + b*x + c = 0
    double a =  94906265.625;
    double b = -189812534.0;
    double c =  94906268.375;

    double delta = b*b - 4*a*c;
    double root1 = (-b + sqrt(delta)) / (2*a);
    double root2 = (-b - sqrt(delta)) / (2*a);

    return root2;
}
```

Code listing 2.2: W. Kahan's second order equation.

The given polynomial has a single root, 1. Computed in float precision, the function returns NaN (not-a-number) because the determinant is evaluated as -2147483648, which is negative, instead of zero due to the round-offs in the multiplications. A more advanced implementation would conclude, wrongly, that the polynomial does not have a real root. In double precision we obtain 1.00000014487 979 and, since we know that this function should output the integer one, we can easily conclude on the precision of the result. This is not generally the case. Long double precision[4] returns 1.000000000000000 which lures us in a false sense of security: one might think that slightly increasing precision is enough to get a correct result.

S. Rump's polynomial

The example given in code listing 2.3 was proposed by Siegfried Rump [Rump, 1988] and implements Equation 2.8.

$$\begin{aligned} x &= 77617 \\ y &= 33096 \\ P(x,y) &= 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + \frac{x}{2y} \end{aligned} \tag{2.8}$$

It was designed to produce the same output with 32 bit, 64 bit, and 128 bit floating-point arithmetics on an IBM, pre IEEE-754 standard, machine. While this

[4]Which is equivalent to 80 bit precision for most C/C++ compilers. The Microsoft Visual C++ compiler is a notable exception, it defines long double as an alias for 64 bit precision.

particular example does not exhibit this behavior on modern processors, it still produces some interesting outputs.

```
double Polynomial()
{
   double x = 77617;
   double y = 33096;
   double x2 = pow(x,2);
   double y2 = pow(y,2);
   double y4 = pow(y,4);
   double y6 = pow(y,6);
   double y8 = pow(y,8);

   double result = 333.75*y6
                 + x2*(11*x2*y2 - y6 - 121*y4 - 2)
                 + 5.5*y8 + x/(2*y);

   return result;
}
```

Code listing 2.3: S. Rump's polynomial.

The function outputs -7.6×10^{30} in float precision but -1.18×10^{21} in double precision (a result with a very different magnitude), and 0.18×10^{21} in long double precision (a similar magnitude but a different sign). Here increasing the working precision, a common heuristic to validate a result (see section 3.1.1), does not increase our confidence in any output. The analytical result is approximately -0.827, orders of magnitude from all previous outputs.

J.M. Muller's sequence

The example given in code listing 2.4 was proposed by Jean-Michel Muller [Muller, 1989], it implements the recurrent sequence given in 2.9. As n increases and with those initial values, the sequence converges to 6 in $\mathbb{R}$.

$$\begin{aligned} U(0) &= 2 \\ U(1) &= -4 \\ U(n+1) &= 111 - \frac{1130}{U(n)} + \frac{3000}{U(n) \times U(n-1)} \end{aligned} \tag{2.9}$$

Given enough iterations, here 200, our implementation converges to 100 in any finite precision.

```
double RecurrentSequence()
{
    double x0 = 2.0;
    double x1 = -4.0;

    for(int i = 1; i <= 200; i++)
    {
        double x2 = 111 - (1130 - 3000/x0)/x1;
        x0 = x1;
        x1 = x2;
    }

    return x1;
}
```

Code listing 2.4: J.M. Muller's sequence.

William Kahan gave an analysis of this example in [Kahan, 2006]. Equation 2.10 gives the general equation of the sequence, where α, β and γ are function of $U(0)$ and $U(1)$. 100 is an attractive point for this sequence: if α has any value different from zero, it will converge to 100. Otherwise, and if $\beta \neq 0$, it converges to 6.

Our initial values, $U(0) = 2.0$ and $U(1) = -4.0$, are chosen such that the sequence converges to 6 ($\alpha = 0$ and $\beta \neq 0$) but the slightest round-off in the computation is enough to make α non-zero which cause the sequence to converge to 100. It is a particularly tricky case as, given enough iterations, no finite precision will be able to reach the analytical result expected from our starting point.

$$U(n) = \frac{\alpha \times 100^{n+1} + \beta \times 6^{n+1} + \gamma \times 5^{n+1}}{\alpha \times 100^{n} + \beta \times 6^{n} + \gamma \times 5^{n}} \tag{2.10}$$

This example is also interesting because the output obtained might not always be considered incorrect. While a mathematician studying the sequence might consider 6 the only acceptable result, a physicist doing simulation may consider 100 to be the correct result if this sequence is used to model a system which is susceptible to receive perturbations not taken into account by the model.

2.4 Error-Free Transformations

When using the rounding-to-nearest rounding mode, it is known [Bohlender et al., 1991] that the numerical error of the addition and multiplication operators in IEEE-754 floating-point arithmetic is a representable floating-point number that can be

computed without needing any additional precision for most[5] floating-points. *Error-free transformations*, as introduced in [Knuth, 1998, section 4.2.2], are operations that, despite being build on floating-point operators, *provably* return the numerical error for any floating-point operands (for a good reference see [Muller et al., 2010, section 4.3, 4.4, 5.1 and 5.2]). They are commonly used to improve the numerical properties of algorithms such as the sum, dot product and polynomial evaluation. Building either *compensated algorithms*, with a reduced numerical error, or *exact algorithms*, guaranteed to be within one ULP of the analytical result. We refer the interested reader to section 6, *Enhanced Floating-Point Sums, Dot products, and Polynomial Values*, of [Muller et al., 2010].

In the following we use two transformations: *TwoSum* [Knuth, 1998, section 4.4.2, theorem B], an error-free transformation able to return the error δ_+ of the addition of two floating-point numbers x and y in the absence of overflow, and a *Fused Multiply-Add* (*fma*) used to extract the error δ_* of their multiplication (see Algorithm 2), but also the exact residual of the division and square root [Muller et al., 2010, section 5.2]. A *Fused Multiply-Add* is an operation which computes $fma(x, y, z) = x * y + z$ with an *infinite* intermediate precision which guarantees that the result will be within 0.5 ULP of the analytical result. Said differently, given some inputs it will produce the representable number that is closest to the analytical output. While it can be implemented at the software level, fast hardware implementations are accessible from most programming languages and for most processors due to the usefulness of the operation, as detailed in [Muller et al., 2010, section 5.2].

Algorithm 1 TwoSum(x, y)

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

Algorithm 2 TwoMultFma(x, y)

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

[5]There are preconditions, which apply to the majority of inputs, to insures that the numerical error of the multiplication operator is a representable number. See [Muller et al., 2010, section 4.4].

One can show that *TwoMultFma* works as expected in a fairly straightforward way. Given a product, $z = \uparrow_n (x * y) = x * y - \delta_*$ with $(z, x, y, \delta_*) \in \mathbb{F}$, where z is the result and δ_* the numerical error we want to extract, we have $\text{fma}(x, y, -z) = \uparrow_n (x * y - z) = \uparrow_n (x * y - (x * y - \delta_*)) = \uparrow_n (\delta_*)$. If we know that the numerical error is a representable number, then $\uparrow_n (\delta_*) = \delta_*$ and thus $\text{fma}(x, y, -z) = \delta_*$.

Twosum cannot be proven to work as easily but one can develop an intuition for why it does work with an example. Setting x to 2^{30} and y to 1 such that the numerical error is equal to one of the inputs, we have:

$$
\begin{aligned}
z &= \uparrow_n (x + y) = \uparrow_n (2^{30} + 1) = 2^{30} \\
x' &= \uparrow_n (z - y) = \uparrow_n (2^{30} - 1) = 2^{30} \\
y' &= \uparrow_n (z - x') = \uparrow_n (2^{30} - 2^{30}) = 0 \\
\delta_x &= \uparrow_n (x - x') = \uparrow_n (2^{30} - 2^{30}) = 0 \\
\delta_y &= \uparrow_n (y - y') = \uparrow_n (1 - 0) = 1 \\
\delta_+ &= \uparrow_n (\delta_x + \delta_y) = \uparrow_n (0 + 1) = 1
\end{aligned}
$$

x' and y' contain the bit that the result z inherited, respectively, from x and y. δ_x and δ_y contain the associated bits lost which, summed into δ_+, give us the numerical error associated with the operation.

It should be noted that, while we focus on the rounding-to-nearest rounding mode, there is a variation of the *TwoSum* error-free transformation that is valid for any rounding mode [Priest, 1992, section 2.3]. There are proofs [Graillat et al., 2009] showing that the *Fused Multiply-Add* operation can still be used as an error free transform for other rounding modes. Using those operations, one could generalize our work to other rounding modes.

Chapter 3

State of the art

Our aim is to develop a method to quantify the impact of the numerical error on the outputs of a given application in a simulation and high-performance computing frame (and not, for example, the certification of single functions in a mathematical library). This chapter does a review of the state of the art through this lens. First, we introduce the approaches that have been used to measure the numerical error of a floating-point computation. Second, we describe the methods that are used to try to localize the sources of error in order to improve the numerical stability of a program.

3.1 Measuring numerical error

This section introduces the classes of methods currently used to assess the numerical error of a program. Their strengths and shortcomings are examined as applied to large simulations and high-performance computing programs.

3.1.1 Comparison with higher-precision arithmetic

The most obvious, and maybe the simplest, way to measure numerical error is to compare an output with the result of an equivalent computation performed with higher precision arithmetic. Higher precision arithmetic is usually obtained with a wider IEEE-754 format, for example 64 bit numbers to analyze 32 bit arithmetic, or with a library implementing arbitrary precision arithmetic, most often the MPFR library [Fousse et al., 2007]. Given two numbers $(x, y) \in \mathbb{F}^2$ and a simple computation, $x + y^2$, the method would be applied as follows[1]:

[1]Where $\{\dots\}_{high\ precision\ computation}$ denotes a computation done with an increased precision.

$$z = x + y^2$$
$$z_{high\ precision} = \{x + y^2\}_{high\ precision\ computation}$$
$$error = z_{high\ precision} - z$$

Good illustrations of this approach include FpDebug [Benz et al., 2012] and Precimonious [Rubio-González et al., 2013]. This is often combined with *Shadow execution* (as in [Benz et al., 2012]): running the program with both precisions at the same time via a form of instrumentation (usually binary instrumentation with a tool such as Valgrind [Nethercote and Seward, 2007]).

While easy to implement (in its simplest form) and to interpret, this has two main limitations: high precision tends to have a large overhead (two orders of magnitude for MPFR in our tests) and this approach relies on the hypothesis that the precision used will be sufficient to detect problems. As we saw in section 2.3.2 and as detailled in section 4.3.4, a large cancellation can make high precision arithmetic unreliable when it is used to estimate the numerical accuracy of a result.

3.1.2 Static analysis

Building on the IEEE-754 standard guarantees on the rounding of the operations, one can use static methods, which are usually built on abstract interpretation (as in FLUCTUAT [Goubault, 2013]), symbolic reasoning (as in FPTaylor [Solovyev et al., 2015]), SMT-solvers (as in Rosa [Darulova and Kuncak, 2014b]), or proof assistants (as in Gappa [de Dinechin et al., 2011]), to prove that a result will have sufficient precision for all possible inputs.

The strength of this type of approach is that it provides strong guarantees for all possible inputs which makes it a method of choice to certify implementation of mathematical functions. However, it is often confined to programs written in domain specific languages and tends to be limited to programs that are both short (often a single function) and simple (loops and tests increase the difficulty of the task considerably) [Darulova and Kuncak, 2014a].

3.1.3 Interval arithmetic

Interval arithmetic [Moore, 1963, Moore, 1966] provides a more scalable way to get strong guarantees on a result. Its key idea is to associate an interval $[x_{inf}, x_{sup}]$ with each number x such that $x \in [x_{inf}, x_{sup}]$ (the interval can also be encoded by a midpoint representation of the form $x \pm x_{width}$, as in Arb [Johansson, 2017], which provides some efficiency benefits). The interval encodes a strict upper bound and a strict lower bound on the result, using the rounding toward $-\infty$ and rounding

toward $+\infty$ to take round-offs into account as seen in Equation 3.1. The output interval gives us an upper bound on the numerical error of the computation.

$$[x^-, x^+] + [y^-, y^+] = [\uparrow_{-\infty}(x^- + y^-), \uparrow_{+\infty}(x^+ + y^+)] \tag{3.1}$$

This approach is of particular interest to developers that want to certify a function or a small program where static methods do not scale or are not applicable. It can even be combined with static analysis to build on the strengths of both methods [Darulova and Kuncak, 2014b]. However, when applied to large computations, these methods tend to return intervals too large to be useful, unless one performs highly specialized modifications of the original computation in order to mitigate the problem (Newton's method famously requires a fix to avoid diverging intervals [Revol, 2003]).

Some work has been done to mitigate the problem, such as affine arithmetic [Comba and Stolfi, 1993] which keeps information on the dependencies between intermediate operations and represents numbers with the form $x_{affine} = x + \sum_i \epsilon_i x_i$ where the x_i are floating-point numbers and the ϵ_i are symbolic values considered to be within $[-1, 1]$. With this formalism, any source of uncertainty (such as round-off error) before or after a computation adds a $\epsilon_i x_i$ term which could later be compensated when combining numbers which share this term (and thus an input). However, no solution currently fully solves the problem of exploding intervals and refactoring an algorithm to use interval arithmetic is still a highly specialized task that might not produce useful results.

3.1.4 Stochastic arithmetic

Stochastic arithmetic [Vignes and La Porte, 1974, Parker, 1997] provides a rather transversal solution to the problem. Its key idea is to add perturbations to every arithmetic operation, either by switching the rounding mode randomly toward $+\infty$ or $-\infty$ (the CESTAC method [Vignes and La Porte, 1974]) or by adding noise to the inputs and output (Monte-Carlo arithmetic [Parker, 1997]), and to run the computation several times while storing the various outputs. The round-to-nearest IEEE-754 result that we want to study, can be seen as a draw from the distribution of outputs. If the distribution has a relatively small standard deviation, then one can conclude that the computation is numerically stable and vice-versa. Furthermore, the mean of this distribution acts as if the arithmetic was associative and should converge, asymptotically, toward the result one would obtain in infinite precision [Chesneaux, 1988].

Given a computation, $(2+3)-4$, the method would be applied as follows[2]:

$$
\begin{aligned}
x_1 &= \{(2+3)-4\}_{stochastic\ arithmetic} = 1.01 \\
x_2 &= \{(2+3)-4\}_{stochastic\ arithmetic} = 0.99 \\
x_3 &= \{(2+3)-4\}_{stochastic\ arithmetic} = 1.05 \\
x_4 &= \{(2+3)-4\}_{stochastic\ arithmetic} = 0.95 \\
\mu_x &= mean(x) = 1.0 \\
\sigma_x &= standardDeviation(x) \approx 0.042 \\
digits(x) &= \left\lfloor -\log_2 \left| \frac{\sigma_x}{\mu_x} \right| \right\rfloor = 4
\end{aligned}
$$

This method, whose notable implementations include Cadna [Jézéquel and Chesneaux, 2008], Verificarlo [Denis et al., 2016] and Verrou [Févotte and Lathuilière, 2016], has been applied successfully to large programs [Knizia et al., 2011] and is particularly suited to applications that already use the Monte-Carlo method (as the applications are already designed to be run several times in parallel).

However, stochastic arithmetic has two drawbacks. First it is very slow[3], one needs to run the operations several times (calling a random number generator for each operation) until a good estimator of the distribution is reached. Second, the output distribution can be complex to interpret. Most implementations of the concept, such as Verificarlo [Denis et al., 2016] and Verrou [Févotte and Lathuilière, 2016], are asynchronous: the users run the program several times, collect the outputs and perform a statistical analysis themselves to conclude on the numerical quality of the result. This is particularly non-trivial for a multi-modal distribution, as can be produced by an unstable test.

A notable exception is Cadna [Jézéquel and Chesneaux, 2008] which encapsulate three synchronous runs of the CESTAC method in a numerical type to avoid re-runs and post-processing (the Discrete Stochastic Arithmetic method [Vignes, 2004]). This makes the statistical analysis automatic and let them detect unstable comparisons on the fly. However, one cannot increase the number of runs to get finer information on the distribution and it uses a majority vote to evaluate comparisons which can impact the control flow of the program (and, for example, cause iterative algorithms to stop sooner than in the uninstrumented application that we want

[2]Where $\{\dots\}_{stochastic\ arithmetic}$ denotes the use of a stochastic arithmetic algorithm to inject noise or switch rounding mode randomly for all operations. The noise has been made artificially large to make the example easier to follow, in practice one would use noise of the order of one ULP.

[3][Jézéquel, 2020] gives a runtime overhead of 5 to 8 for Cadna, about 30 for Verrou and between 300 and 600 for Verificarlo. We do further benchmarks on arithmetically dense computations in section 7.1.

to validate[4]). As detailed later, parts of our interface and the idea of using an instrumented numerical type in our work are inherited from the design of Cadna.

3.1.5 Local error measuring

Some recent publications, such as [Zou et al., 2019] and [Bao and Zhang, 2013], have explored entirely local analysis. While there are variations, their common idea is to measure an error metric for individual operations without taking into account any error propagation from one operation to the other. Typically, they measure the loss of magnitude of the outputs of arithmetic operators which can denote catastrophic cancellations. When the metric goes above a given threshold, the operation is reported. If enough operations have been reported, then the computation is considered numerically unstable.

One advantage of this approach is that one can instrument operations without modifying the memory representation of numbers which can be done fairly easily with operator overloading or binary instrumentation (with a tool like Valgrind). These methods appear to work well on small, single functions, where one can work under the hypothesis that the number of operations is small and that the output is impacted by all operations.

However, their entirely local nature is a shortcoming when dealing with very large programs. This approach can fail to detect slow accumulations of numerical errors (introducing false negatives), while flagging any program which has a locally unstable component, even if this component does not impact the final result (a false positive).

3.2 Localizing the sources of numerical error

Once one can measure the numerical error of a program and determine that there is a problem, the next logical step is to try to localize the sources of error. However, as solving this problem requires the ability to measure the numerical error, fewer algorithms have been developed to tackle it. This section covers the two dominant approaches for finding the sources of error.

3.2.1 Local error reporting

A fairly straightforward solution to identify the sources of error in a program is to find the operations where the largest errors occur. This is a logical outgrow for the error measuring algorithms based on local analysis (as presented in section

[4]This property can be a good thing if one wants to make a stopping criteria more stable.

3.1.5); it is also implemented in Cadna [Jézéquel and Chesneaux, 2008] with their numerical debugger which is triggered whenever a large cancellation occurs.

In its more straightforward form, this method is fairly simple to implement as it is entirely local, but it shares the downsides of the other purely local approaches to numerical error. It detects single large cancellations (or sensible operations), even if they have no impact on the output of the problem, but is oblivious to a slow accumulation of numerical error making it an unreliable approach to analyze medium to large programs.

Refinements exist, such as [Zou et al., 2019], that follow the canceled bit to see whether it impacts the final output. While they help to deal with false positives, they still rely on the hypothesis that no slow accumulation of error will happen (hypothesis that is made explicit in [Zou et al., 2019, section 6.1.I3]).

3.2.2 Delta-Debugging

When one has an algorithm to measure the numerical error of a program, an option to identify the sources of error might be to deactivate the measure locally and find a subset of operations that suffice to reproduce the output error. As this is a combinatorial problem, one needs an heuristic to prune the search space. One solution, introduced by Precimonius [Rubio-González et al., 2013] and used by Verrou [Févotte and Lathuilière, 2016], is to use Delta-Debugging [Zeller, 2009], an algorithm usually used to locate bugs.

Delta-Debugging can be used to locate the operations of interest by binary search on the source code. It requires a way to set the error produced by one or more areas of the program to zero and a binary function to classify an output as correct or incorrect (this function is subject to tweaking in order to produce an informative output).

The main downside of this approach is that, despite the huge speed-ups introduced by the use of Delta-debugging over a naive search, it still requires numerous instrumented runs of the program to zone in on the sources of error, making it very slow. Furthermore, it provides relatively crude information: for example it can flag two functions as major sources of error but will not be able to warn the user that one of those functions produces ten time more numerical error than the other.

3.3 Conclusion

It seems to us that there is a lack of methods to measure numerical error and localize its sources that can scale to very large programs and high-performance computing. The problem is that one needs a method that has an overhead low enough to be applied to a code that already takes a long time to run in its uninstrumented

form and that stays accurate when confronted with a large number of operations. Furthermore, such a method should be able to deal with parallelism and varied operations, as any program large enough can be expected to have at least one call to an uncommon mathematical function from the standard library.

With our work, we offer a method with distinct properties: both fast enough to be used on large simulations and providing easily understandable results that can be directly used and interpreted by end-users. To do so, we observe that part of the work that has been done to increase the precision of computation, most notably pair arithmetic [Lange and Rump, 2020] (which derives from double-double arithmetic [Dekker, 1971, Bailey, 1995]), can be repurposed and extended to model numerical error on the fly. Furthermore, it deals correctly with the cancellation problem that makes high precision arithmetic an unwise choice to measure numerical error. To the best of our knowledge, we are the first to adapt those concepts to the measure of numerical error.

Part II

Encapsulated error: A direct approach to assess floating-point accuracy

Table of Contents

Chapter 4

Algorithms

Having identified a lack of methods suited for the analysis of large application and high-performance computing, we developed two methods based on number representations that encapsulate additional information.

In the first section of this chapter, we introduce the concept of *encapsulated error*, which we developed to measure the numerical error through a computation, and detail the algorithms needed to implement it. In the second section we present an extension of our method, *tagged error*, to not only measure numerical error but also trace it to its origins. Finally, we analyze some key characteristics of the method.

4.1 Encapsulated error

We propose the use of what we call *encapsulated error*, to assess floating-point accuracy. Our idea is to replace floating-point numbers with a pair $(number, error)$ which contains both the result of the original IEEE-754 floating-point computations ($number$), and a *signed* first-order approximation of its current numerical error ($error$), with respect to the machine precision, such that their sum ($number + error$) is a first-order approximation of the result we would have obtained in infinite precision.

The folllowing sections describe our method and, in particular, the way we apply operations to numbers, compare them, and display them.

4.1.1 Arithmetic operators

All operations on our type, arithmetic operators ($+$, $-$, $\times$, $/$) as well as arbitrary function, take $(number, error)$ pairs, such as (x, δ_x) and (y, δ_y), as inputs and output a (z, δ_z) pair, where z is the result of the operation and δ_z its numerical

error. δ_z is computed using both the error introduced by the operation and the error transmitted from the inputs (δ_x and δ_y).

When we evaluate arithmetic operators and the square root function, we compute the error produced by the operation using an error-free transformation (explained in section 2.4) and combine it with the error transmitted from the inputs using basic arithmetic:

Algorithm 3 Addition((x, δ_x), (y, δ_y))

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

Algorithm 4 Multiplication((x, δ_x), (y, δ_y))

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

Algorithm 5 Division((x, δ_x), (y, δ_y))

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

Algorithm 6 Square Root((x, δ_x))

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

The error in the multiplication operation should be $(\delta_x * y) + (\delta_y * x) + (\delta_x * \delta_y) + \text{fma}(x, y, -z)$ but we decided to omit the second order term, $\delta_x * \delta_y$, from the computation of the error and to linearize the square root using its first-order Taylor approximation. See Section 4.3.2 for an analysis of the associated trade-off.

The previous algorithms are equivalent to double-double arithmetic [Dekker, 1971, Bailey, 1995] without the final re-normalization step and have been introduced independently in [Latkin, 2014] and [Lange and Rump, 2020]. Our work is different in that it gives a different semantic to this representation: our pairs model numbers and their errors rather than higher precision numbers. This means that the error term should *never* impact the value of the numbers (hence the absence of the re-normalization step) or comparisons and the control flow of a computation.

As detailed in section 4.3.3, the error bounds from [Lange and Rump, 2020] apply to our arithmetic and can be used to prove that, under their hypothesis and if we restrict ourselves to basic arithmetic operations, our computation of the numerical error is numerically stable. We test the accuracy of our evaluation on a practical case in chapter 6.

4.1.2 Arbitrary functions

For arbitrary functions, without any easy way to estimate the error introduced by the function, we deduce the final error by using higher precision arithmetic[1] to subtract the IEEE-754 arithmetic result from a higher precision result computed after having corrected our number with its error:

Algorithm 7 Arbitrary Function(f, (x, δ_x))

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

4.1.3 Comparisons

In order to preserve the control flow of the original IEEE-754 computation, comparisons and tests are performed on the *number* part of the pair. Thus, given two floating-point numbers $(x, y) \in \mathbb{F}^2$ and the associated numerical error approximations $(\delta_x, \delta_y) \in \mathbb{F}^2$, the comparison of the pairs is equivalent to the comparison of the numbers:

$$(x, \delta_x) \leq (y, \delta_y) \Leftrightarrow x \leq y$$

Whenever we call a comparison operator, we can raise a warning if the subtraction of its arguments produces a number with no significant digits (which Jean Vignes calls a "zéro informatique" [Vignes and ARSAC, 1986]). It thus means that the comparison is numerically unstable (this approach is inspired by the implementation of Cadna [Jézéquel and Chesneaux, 2008]).

[1] In practice, we use twice the working precision.

4.1.4 Output

We display $(number, error)$ pairs with algorithm 8. Our algorithm requires the function `digits` that computes the number of significant digits of a $(number, error)$ pair (introduced in section 2.2.3), a function `print_string` that prints an arbitrary string of characters and a function `print_scientific` that displays a number in scientific notation with a given number of significant digits.

Algorithm 8 Display$((number, error))$

$digits\_number \leftarrow digits(number, error)$
$digits\_number_0 \leftarrow \lfloor -\log_{10} |error| - 1 \rfloor$
if $digits\_number > 0$ **then**
  return *print_scientific(number, digits_number)*
if $(digits\_number_0 > 0)$ **and** $(|number| < 1)$ **then**
  return *print_scientific*$(0, digits\_number_0)$
return *print*("∼numerical-noise∼")

When a result has significant digits (according to our error measure), we display it in scientific notation with the computed number of significant digits. When a result has no significant digits, but is lower than one in absolute value, we compute what we call its number of significant zeroes ($\lfloor -\log_{10} |error| - 1 \rfloor$), and, if it is positive, display zero with that number of zeroes. Otherwise, we print ∼`numerical-noise`∼ which is our way to signify that a result does not contain meaningful information.

This algorithm, and in particular the idea of significant zeroes, is inspired by the display function used by Cadna [Jézéquel and Chesneaux, 2008] (whose details are, to our knowledge, undocumented). Significant zeros let us encode $(10^{-7}, 10^{-6})$ as 0.0000, preserving the magnitude information which would otherwise be discarded if we displayed it as ∼`numerical-noise`∼.

4.2 Tagged Error

Once one has the ability to measure numerical error and detect that a given computation is affected by a large numerical error, the next logical step is to find the sources of the numerical error in order to try to mitigate it.

We propose an extension of our method, called *tagged error*, with one error term $(error_i)$ per user-defined section of the code instead of a single term. This extension, inspired by affine arithmetic [Comba and Stolfi, 1993], lets us track the evolution, amplification or dampening, of the numerical error produced by various code sections independently of one another through the rest of the computation.

This section details the modifications to the previous algorithms required to introduce tagged error. We define *tags*, operations on our numbers and the way we display the tracking information.

4.2.1 Tags and sections

We call *section* a line or group of lines in a source code that all fall within the same scope. We call *tag* a user-defined name associated to a section that will be associated with an error term.

A number is represented as a pair $(number, [error_i])$ where $[error_i]$ is a mapping between each tag i and the value of their associated error terms. In practice, this mapping can be implemented with an array as long as the tags i are consecutive integers.

We define a function `CurrentTag` which can be used to get the tag associated with the current section of the code.

4.2.2 Arithmetic operators

Our operations build on the operation used in *encapsulated error*, they take (*number*, $[error_i]$) pairs, such as $(x, [\delta_{xi}])$ and $(y, [\delta_{yi}])$, as inputs and output a $(z, [\delta_{zi}])$ pair where z is the result of the operation and the $[\delta_{zi}]$ its numerical error terms. A term δ_{zi} is computed using the error transmitted from the inputs (δ_{xi} and δ_{yi}) and, optionally, the error introduced by the operation. The two main differences are the fact that we manipulate not one error term, but an array of error terms and that the error introduced by the operation is considered to be zero for all but the tag corresponding to the code section in which the operation was done.

Algorithm 9 Addition($(x, [\delta_{xi}])$, $(y, [\delta_{yi}])$)

$z \leftarrow x + y$
for i **in** tags **do**
    $\epsilon_i \leftarrow$ **if** CurrentTag() $== i$ **then** TwoSum(x, y) **else** 0.0
    $\delta_{zi} \leftarrow \delta_{xi} + \delta_{yi} + \epsilon_i$
return $(z, [\delta_{zi}])$

Algorithm 10 Multiplication($(x, [\delta_{xi}])$, $(y, [\delta_{yi}])$)

$z \leftarrow x * y$
for i **in** tags **do**
  $\epsilon_i \leftarrow$ **if** CurrentTag() == i **then** fma$(x, y, -z)$ **else** 0.0
  $\delta_{zi} \leftarrow (\delta_{xi} * y) + (\delta_{yi} * x) + \epsilon_i$
return $(z, [\delta_{zi}])$

Algorithm 11 Division($(x, [\delta_{xi}])$, $(y, [\delta_{yi}])$)

$z \leftarrow x/y$
for i **in** tags **do**
  $\epsilon_i \leftarrow$ **if** CurrentTag() == i **then** fma$(y, z, -x)$ **else** 0.0
  $numerator_i \leftarrow (\delta_{xi} - \epsilon_i) - z * \delta_{yi}$
  $denominator_i \leftarrow y + \sum_j \delta_{yj}$
  $\delta_{zi} \leftarrow numerator_i / denominator_i$
return $(z, [\delta_{zi}])$

Algorithm 12 Square Root($(x, [\delta_{xi}])$)

$z \leftarrow \sqrt{x}$
for i **in** tags **do**
  $\epsilon_i \leftarrow$ **if** CurrentTag() == i **then** fma$(-z, z, x)$ **else** 0.0
  $numerator_i \leftarrow \delta_{xi} + \epsilon_i$
  $denominator_i \leftarrow z + z$
  $\delta_{zi} \leftarrow numerator_i / denominator_i$
return $(z, [\delta_{zi}])$

4.2.3 Arbitrary functions

For arbitrary functions, the error introduced by the function (which will be stored in the current tag) is computed using higher precision arithmetic. The impact of the error propagated from the inputs is also computed with higher precision arithmetic before being distributed proportionally between the error terms (this is equivalent to a linear approximation).

Algorithm 13 Arbitrary Function(f, $(x, [\delta_{xi}])$)

$z \leftarrow f(x)$
$z_{precise} \leftarrow \{f(x)\}_{high\ precision\ computation}$
$z_{corrected} \leftarrow \{f(x + \sum_j \delta_{xj})\}_{high\ precision\ computation}$
for i **in** tags **do**
    $\epsilon_i \leftarrow$ **if** CurrentTag() == i **then** $z_{precise} - z$ **else** 0.0
    $\delta_{zi} \leftarrow \frac{\delta_{xi}}{\sum_j \delta_{xj}} * (z_{corrected} - z_{precise}) + \epsilon_i$
return $(z, [\delta_{zi}])$

4.2.4 Output

If the number 1.2341152 has a total numerical error of 0.003 split between five tags ($func1 = 0.0027$, $func2 = -0.0006$, $func4 = -0.0001$, $func5 = -0.0001$, $func6 = -0.0001$), it would be represented as follows:

$$1.23\ [func1 : 90\%, func2 : -20\%, ...]$$

We display the number first, only the digits considered significant according to the total numerical error, followed by a list of error terms with their tag name in order of decreasing impact, each associated to a signed percentage of the total error. Displaying signed quantities make it possible to represent compensation between various code sections.

Tags with a numerical error of exactly zero are omitted. If some tags have a value that is non-null, but less than 5%, they are omitted and trailing dots are added to the display to denote their existence.

4.3 Discussion on encapsulated error's properties

This section tries to answer to some theoretical questions and pinpoint the specificities of the method. We detail the characteristics that make our method unique, the reason that led to using a first order approximation, the error bound one can expect on our approximation of the numerical error and the fundamental differences between our method and higher precision arithmetic.

4.3.1 Characteristics

Our method was built to have several key characteristics:

- First and foremost, it can give us an estimation of the *numerical error* of any number in a given computation. Thus, a user can easily analyze a program, step by step, if needed.

- Second, our outputs have to be pertinent for the non-instrumented computation. Meaning that the instrumented code should follow the same branches and code paths as the non-instrumented code and that the user should be able to confirm that the input was left unchanged by the instrumentation.

- Third, we need to keep the runtime overhead low enough to ensure that our method can be applied to very large simulations.

Having those design criteria in mind, it is interesting to review some characteristics of the resulting method. As the comparisons are computed on the *number* part of the pair, our method ensures that we go through the same path and branches as the original computation. Therefore, it gives us both the same result that we would have had with the original computation in IEEE-754 arithmetic[2] and a first-order approximation of its numerical error. Having both quantities for all numbers, it is thus possible to get the amplitude of the numerical error and an estimate of the number of significant digits at all times and for all intermediate results.

Furthermore, implementations can test the quality of the numbers produced after each operation and forward the information to a debugger in order to pinpoint the origin of the numerical inaccuracies in the computation (a functionality inspired by Cadna [Jézéquel and Chesneaux, 2008]).

However, it is essential to note that, contrary to asynchronous methods such as stochastic arithmetic, the control flow of the computation is not altered. While this ensures that our analysis is pertinent to the outputs of the non-instrumented computation, this means that we only explore and return information about the branches followed by the original computation. A more accurate result might have followed different branches with vastly different behaviors due to unstable comparisons; we know nothing of those behaviors. In practice, this has been observed to cause our method to underestimate the precision of some algorithms designed to be resilient to numerical errors in intermediate steps. Hence when validating an output, one needs to look at the estimation of the number of significant digits but also insure that meaningful tests are stable. Here, our method's ability to detect unstable branches is primordial.

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

Finally, our method is compatible with parallelism and, as we will present in chapter 7, has a low overhead compared to the state of the art. It makes it a suitable candidate to analyze high-performance computations.

4.3.2 Second-order term

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

If the computation of a number x is numerically stable then, by definition, given the machine epsilon ϵ (2^{-53} for 64 bit double precision), its error is of order $o(|x| * \epsilon)$. Adding a second-order term to our estimate adds a correction of order $o(|x| * \epsilon^2)$, but the error introduced by the addition of the correction itself is of order $o((|x| * \epsilon + |x| * \epsilon^2) * \epsilon) = o(|x| * \epsilon^2 * (1 + \epsilon))$ which could dominate the correction creating spurious results.

Meanwhile, if the error is larger than $o(|x| * \epsilon)$, then the first-order approximation of the numerical error reflects the fact that the computation is not numerically stable without requiring second-order terms.

Note that it should be possible to integrate this second order term in a numerically stable way with an alternative formula based on a compensated dot product. However, the impact on the computing time would be major while, according to our tests, the accuracy would not improve sensibly.

4.3.3 Error bounds

The error bounds computed in [Lange and Rump, 2020] apply to our arithmetic as long as we restrict ourselves to basic arithmetic operations (avoiding arbitrary functions which are not covered by [Lange and Rump, 2020]).

Given an arithmetic expression defined by an evaluation tree T, where each node n is associated with an operation o_n out of $\{+, \times, /, \sqrt{}\}$, we can assign a number k_n to each node such that:

$$k_n = \begin{cases} 0 & \text{if } n \text{ is a leaf} \\ max(k_{\text{left}(n)}, k_{\text{right}(n)}) + 1 & \text{if } o_n \text{ is a } + \\ k_{\text{left}(n)} + k_{\text{right}(n)} + 1 & \text{if } o_n \text{ is a } \times \\ k_{\text{left}(n)} + k_{\text{right}(n)} + 2 & \text{if } o_n \text{ is a } / \\ \lceil \frac{4}{5} k_{\text{child}(n)} + \frac{5}{4} \rceil & \text{if } o_n \text{ is a } \sqrt{} \end{cases}$$

We define:

- $u = \frac{1}{2}b^{1-m}$, the error constant associated to an m digits floating-point system with base b (u being 2^{-53} for 64 bit double precision),
- x and δ_x, the number and error approximation obtained when we evaluate an evaluation tree T_x.
- $k = k_{\text{root}(T_x)}$, the positive integer which is associated with the root of the evaluation tree that produces x,
- $\widehat{x}$, the result that would be obtained in infinite precision,
- $\widehat{X}$, the positive number that would be obtained after transforming T_x such that it respects the *No Inaccurate Cancellation* (NIC) principle from [Demmel et al., 2008], To respect the NIC principle, one needs to modify the signs of intermediate results so that numbers of different signs are never added together unless they are inputs to the algorithm.

Given those definitions and under the hypothesis that all denominators and all expressions below a square root comply with the NIC principle and that the nodes corresponding to divisions and square root operations have a k_n smaller than $u^{-\frac{1}{2}}$, theorem 4.2 from [Lange and Rump, 2020] gives us an upper bound of the error of our approximation of the numerical error:

$$|\widehat{x} - (x + \delta_x)| \leq k \times (k+2) \times (1+2u)^k \times u^2 \times \widehat{X}$$

4.3.4 Comparison with higher precision arithmetic

One might wonder how could this scheme be more precise than deducing the error from doing the computation with our target precision and higher precision arithmetic side by side. Especially since the usual rule of thumb for compensated summation and arithmetic relying on error-free transformations is to assume that they are equivalent to doubled precision [Thévenoux et al., 2015].

For most operations you do, indeed, get the same error estimation as you would with roughly doubled precision (minus any overlapping bit between the value of the number and its error). However, the decisive difference is in the resilience to cancellations. For instance, both 64 bit floating-point arithmetic and 200 bit floating-points arithmetic would evaluate $(2^{200} + 1) - 2^{200}$ as 0 (since, even with 200 bit of precision, $2^{200} + 1$ rounds to 2^{200}). A comparison between both results would conclude that 0 is indeed the proper result and that there appear to be no numerical error. However, by keeping the error as a separate quantity, the error

introduced during the cancellation (1) is kept separated from the numbers that caused it and, thus, the error is still properly evaluated:

$$(\{2^{200}, 0\} + \{1, 0\}) - \{2^{200}, 0\} = \{2^{200}, 1\} - \{2^{200}, 0\} = \{0, 1\}$$

It is important to note that our approach is optimized to keep track of the error and *not* to increase the precision of the computation. We could increase the precision of our implementation by minimizing the number of overlapping bit between the two terms of the pair (the result would be similar to double-double arithmetic [Dekker, 1971]) but they would not represent the number obtained without instrumentation and its numerical error. This would force us to run the computation a second time to be able to evaluate the numerical error.

By keeping a strict separation between the number and its numerical error, we make the computation of the numerical error efficient in terms of both number of operations and memory.

Chapter 5

Implementation

One of the strengths of the method is the simplicity of the core algorithms: the operations are independent, can be written in a few lines of codes requiring only basic arithmetic operators and a *Fused Multiply-Add* in the working precision. It makes writing a correct implementation easy. Most of the work (and of the implementation) is focused on the ergonomy of the library. We build on operator overloading, having the programming language replace all numerical operators and functions by our implementations, to make the instrumentation seamless, and cover a wide range of operations and functions.

In this chapter we describe our optimized C++ reference implementation [Demeure and Chevalier, 2019], our Julia implementation [Demeure and Ancellin, 2020], the design of our implementation of tagged error, and the tools we designed to track the numerical error and instrument a program.

5.1 Encapsulated error implementation

5.1.1 C++

Our C++ reference implementation, the Shaman library [Demeure and Chevalier, 2019], relies on a custom numeric type with overloaded operations to instrument every arithmetic operation and function call. We decided to implement it such that, if one does not need tagged error (which require some data structure initialization), the library is *header-only* and thus can be dropped into a code base and used without any installation or dependencies.

Our custom type, `S<numberType, errorType, preciseType>`, uses the C++ template system to build an instrumented type on top of any IEEE-754 compatible type. It takes three arguments, `numberType` which is the IEEE-754 compatible type that will be emulated, `errorType` which is the precision at which the error terms

will be manipulated and stored (usually identical to `numberType`) and `preciseType` which is the precision used when higher precision is required (to deal with arbitrary functions). Hence, we can define `Sdouble = S<double, double, long double>` to indicate that we want to use an instrumented type that behaves like the *double* type that is commonly used in C and C++ (numerical errors and implicit casts will be those of the *double* type), stores and manipulates numerical errors in a *double* and does its higher precision computations using the *long double* type (which is only used when computing an arbitrary function such as a sine).

We define the `Sfloat`, `Sdouble` and `Slong_double` types out of the box but our implementation can be used with any user-defined numerical type, as long as it rounds to nearest according to the IEEE-754 standard. An example would be emulated 16-bit precision types (we, in fact, provide an `SFloat16` type out of the box in our Julia implementation). Furthermore, the user can use any type to manipulate the numerical error (it can be used to experiment with further levels of instrumentation such as using Shaman to measure the numerical error in shaman's estimation of the numerical error) and do higher precision computations.

Following our method to instrument arbitrary functions (algorithm 7 page 46), our implementation covers the 73 mathematical functions from the current C++ standard library. We also provided an implementation of the C++ streaming operator which prints number in scientific notation following the algorithm given in Section 4.1.4. This is our preferred format to assess a result.

The most complex part of the implementation is the definition of the mixed-precision operation overloads. We needed to ensure that arithmetic operators would use the proper conversions for all pair of input types (thus an addition between a `double` and a `float` should be done in `double` precision). This can be done with careful code duplication, introducing sixteen variant per operator, but we decided to use a mix of C style macro (to handle the code generation) and template (to extract the proper working precision programmatically) to achieve this functionality. While this produce a hundred lines of complex code, it insures that we use the same type conversion as the one used by the instrumented types and keeps our implementation short by encapsulating the functionality.

Integrating our number representation within libraries and framework can be done at the user level as there is no protected lower level of abstraction within Shaman. For example, having a seamless integration between Shaman and OpenMP [Dagum and Menon, 1998] only required implementing the reduce operations for our number representations which can be done in one line for each (operation, type) pair as seen in code listing 5.1.

```
#pragma omp declare reduction(+:Sdouble : omp_out=omp_in
  +omp_out) initializer(omp_priv=Sdouble(0.0))
```

Code listing 5.1: Implementation of an OpenMP reduce operation.

We also made our implementation of encapsulated error compatible with MPI [Gropp et al., 1996] by defining the `MPI_SFLOAT`, `MPI_SDOUBLE` and `MPI_SLONG_DOUBLE` types (which describe the memory layout of our numbers) and the `MPI_SMAX`, `MPI_SMIN`, `MPI_SSUM` and `MPI_SPROD` operations (which use our overloaded operators).

Finally, we implemented type traits so that our numbers can be used transparently within the Eigen [Guennebaud et al., 2010] and Trilinos [Heroux et al., 2005] linear algebra libraries. Those traits describe the properties of our numbers (whether they are integers, complex, etc) and some operations (how to get the epsilon machine, how to compare two numbers, etc) and were derived from the trait of the instrumented types such that, for example, `Sfloat = S<float, float, double>` would behave like `float`.

5.1.2 Julia

We also provide a Julia reference implementation [Demeure and Ancellin, 2020]. Julia [Bezanson et al., 2017] is a programming language with a strong focus on dynamic typing and numerical computing. As such, most Julia libraries are designed to be generic with regard to the numeric type of their inputs. This makes Julia a perfect fit for type based instrumentation, the language has already been used to explore domains such as uncertainty quantification [Giordano, 2016] and automatic differentiation [Revels et al., 2016] with similar approaches. Furthermore, Julia uses a Just-In-Time compiler which produce code heavily optimized for the current types via LLVM [Lattner, 2008], insuring performances close to C++ if one is careful to insure type stability within functions.

While our Julia library is a fairly straightforward implementation of the previous algorithms (providing `SFloat16`, `SFloat32`, `SFloat64` and `SFloat128` types out of the box), we benefit from the ecosystem which was designed with type based instrumentation in mind and provides us with functionalities such as the `promote_rule` function which can be used to easily specify the type conversion to be used in mixed-precision operations (a complex part of our C++ implementation). Thanks to their focus on dynamic typing, the majority of Julia libraries (including BLAS implementations, Fast Fourier Transforms and a large panel of numerical algorithms) can be instrumented with Shaman by simply feeding instrumented inputs to them.

However, as our Julia implementation has yet to be tested on a large program, the following deals only with the C++ implementation.

5.2 Tagged error implementation

In practice, whenever tagged error is enabled, we represent numbers as triplet $(number, error, [error_i])$. This representation lets us easily disable the tagged error at compile time and gives us access to the total error without having to compute the, potentially numerically unstable, sum of the individual error terms.

5.2.1 Section delimitation and tag retrieval

A section of the code is represented by a tag which is a user-defined name and an integer (used to index into various arrays). The mapping between names and integers is done both ways using a hash-table and a resizable array (commonly called a vector in C++).

To delimitate sections, the user has access to a macro which, when included in the code, will start a section with a given name and close it at the end of the current scope (as will be illustrated in section 5.3.2). If the user wants to use a single section for a function, he just needs to include the macro at the beginning of the function. If the user wants a line by line granularity, he just needs to insert several macros, one before each line of code. In practice, we recommend running the code several times while increasing the granularity, focussing on the sources of error.

Behind the scene, the macro creates a `CodeBlock` object which is destroyed at the end of the scope following the *Resource Acquisition Is Initialization* (RAII) principle common in C++ and Rust. The constructor of the object requires a tag name and converts it into an index by querying it with the corresponding hash-table (and creating it, if needed). The index is then pushed onto a stack that keeps track of the current tag. When the object is destroyed, the stack is popped which removes its tag. This relies on the hypothesis that tags are destroyed in reverse order of their creation which is ensured by the deterministic destruction provided by RAII.

To this are added a `main` tag (which ensures that the stack is never empty) and a `compileTime` tag (which is used to deal with any error introduced by operations that happen before the program is run). With `main` covering the operations that are not within user-named sections, we ensure that all the operations have an associated section and that no numerical error goes unaccounted for.

All of those datastructures and functions are illustrated with pseudocode in Listing 5.2.

```
name_of_index = Vector(["compileTime", "main"])
index_of_name = HashTable(["compileTime"->0, "main"->1])
stack = Stack([0,1])

function CurrentTag():
    return stack.top()

function NameOfTag(index):
    return name_of_index[index]

object CodeBlock:
    function constructor(name):
        if not (name in index_of_name):
            index = name_of_index.length()
            name_of_index.push(name)
            index_of_name[name] = index
        index = index_of_name[name]
        stack.push(index)

    function destructor():
        stack.pop()
```

Code listing 5.2: Pseudo code illustrating the section and tag management.

5.2.2 Operations

Some old C++ code bases create arrays using `malloc` instead of `new[]` which means that their content is not initialized. This leads to problems when an object that points to another resource is accessed or implicitly destroyed to insert a properly constructed object in the array. To avoid those problems the error terms, associated with the tags, are stored in fixed size arrays. This requires knowing the number of tags that will be used in advance but, while this imposes a small constraint to the user (setting a compile time parameter to a value equal or higher to the number of tags that will be used) which would be avoided by using dynamically sized vectors, it let us represent our numbers with was is called a *Plain Old Data* (POD) class in C++ which, in our tests, significantly improves performances.

While the algorithms in section 4.2.2 uses a loop and a test to check whether an operation happens in the current tag, in practice we do array wise operations followed by a targeted modification of the cell representing the current tag. Using fixed size arrays and array-wise operations lets the compiler vectorize the code and

improves performances significantly.

```
#include <iostream>
#include <shaman.h>

int main()
{
 Sdouble x = 2.0;
 Sdouble y = 3.0;

 Sdouble z = x + y;

 return 0;
}
```

```
.....
vmovsd (%rdx,%rdi,1),%xmm2
vaddsd (%rsi,%rdi,1),%xmm2,%xmm2
vmovsd %xmm2,0x8(%rcx,%rdi,1)
vmovsd 0x8(%rdx,%rdi,1),%xmm2
vaddsd 0x8(%rsi,%rdi,1),%xmm2,%xmm2
vmovsd %xmm2,0x10(%rcx,%rdi,1)
vmovsd 0x10(%rdx,%rdi,1),%xmm2
vaddsd 0x10(%rsi,%rdi,1),%xmm2,%xmm2
vmovsd %xmm2,0x18(%rcx,%rdi,1)
vmovsd 0x18(%rdx,%rdi,1),%xmm2
vaddsd 0x18(%rsi,%rdi,1),%xmm2,%xmm2
.....
```

Code listing 5.3: Code instrumented with Shaman and extract of the corresponding assembly side to side.

The vectorization can be observed in the assembly corresponding to code listing 5.3. We compiled the code, an addition done with tagged error enabled and 100 tags, with the Clang++ compiler and the `-mfma -O3 -march=native -Rpass=vect` optimization flags on an Intel(R) Xeon(R) CPU E3-1220 v3. Due to the `-Rpass=vect` option, the compiler notified us that it vectorized the transform operation used to sum two vectors containing error terms. Looking at an extract of the assembly for the main function (exported in the elf64-x86-64 format), we indeed see a large number of `vaddsd` and `vmovsd` instructions, AVX vectorized addition and move operations, which is consistent with the fact that we are adding numbers into a new array.

5.3 Usage

This section illustrates the use of both encapsulated error and tagged error for the analysis of Heron's square root algorithm.

5.3.1 Encapsulated error

Code listing 5.4 gives an example of Heron's square root algorithm implemented in C++ and instrumented with Shaman. Note that the only modification is the change in type, *double* becoming *Sdouble*, which gives us the ability to access both the number and its error.

```
// Sdouble definition:
// using Sdouble = S<double double, long double>;

Sdouble heron(Sdouble x)
{
    Sdouble r = x/2;
    int i = 0;

    while(1e-15 < abs(r*r - x))
    {
       r = (r + x/r) / 2;
       i++;
       std::cout << "iteration:" << i << ' '
                 // displays only significant digits
                 << "sqrt:" << r << std::endl
                 // direct access to the number
                 << "number:" << r.number << std::endl
                 // direct access to the error
                 << "error:" << r.error << std::endl
                 << std::endl;
    }

    return r;
}

void main()
{
    heron(2);
}
```

Code listing 5.4: C++ implementation of Heron's square root algorithm instrumented with encapsulated error.

We output both the result with the overloaded C++ streaming operator, which only prints significant digits, and the number and error approximation parts of our representation. When executed with the numerical debugger enabled in order to get a summary of the instability at the end of the execution, the code prints code listing 5.5.

```
iteration:1 sqrt:1.5000000000000000e+00
number:1.500000000000000e+00
error:-0.000000000000000e+00

iteration:2 sqrt:1.41666666666667e+00
number:1.416666666666667e+00
error:1.480297366166875e-16

iteration:3 sqrt:1.414215686274510e+00
number:1.414215686274510e+00
error:1.393221050510000e-16

iteration:4 sqrt:1.414213562374690e+00
number:1.414213562374690e+00
error:4.079910214529664e-17

iteration:5 sqrt:1.414213562373095e+00
number:1.414213562373095e+00
error:1.253716726897950e-16

*** SHAMAN ***
There are 5 numerical instabilities
3 CANCELLATION(S)
0 UNSTABLE DIVISION(S)
0 UNSTABLE MULTIPLICATION(S)
0 UNSTABLE MATHEMATICAL FUNCTION(S)
0 UNSTABLE POWER FUNCTION(S)
2 UNSTABLE BRANCHING(S)
```

Code listing 5.5: Output for the implementation of Heron's square root algorithm instrumented with encapsulated error.

Note that here the numerical error is small enough to keep the number of significant digits high. As the output does not tell us when and where the cancellations or the unstable branches occur, we use a debugger to pause the computations by having breakpoints on cancellations or unstable branches. Doing so, we learn that from the iteration 4 onward the subtraction in the loop condition is a cancellation. We also learn that the test and the computation of the absolute value become unstable on the last iteration. The localization of the various instabilities can also be seen in the output of the numerical profiler (code listing 5.6, the line 9 corresponds to the loop condition of the code which is placed inside the *heron*

function of the *main.cpp* file).

```
*** SHAMAN PROFILE ***
5       heron (file main.cpp)
3               operator- (line 9)
1               operator<  (line 9)
1               abs (line 9)
```

Code listing 5.6: Output of the numerical profiler.

Hopefully, this example illustrates that, with minimal modifications (a change in types), Shaman gives us access to the numerical error with very fine granularity.

5.3.2 Tagged error

In code listing 5.7, we refactored the formula from the previous example to have two distinct operations within the loop (the computation of a correction and its application to the previous value) and added markers to delimitate various sections of interest.

```
1  Sdouble heron(Sdouble x)
2  {
3      LOCAL_BLOCK("init");
4      Sdouble r = x/2;
5
6      while(1e-15 < abs(r*r - x))
7      {
8          LOCAL_BLOCK("delta");
9          Sdouble delta = 0.5*(x/r - r);
10
11         LOCAL_BLOCK("add");
12         r += delta;
13
14         std::cout << "sqrt:" << r << std::endl;
15     }
16
17     return r;
18 }
19
20 void main()
21 {
22     heron(2);
23 }
```

Code listing 5.7: C++ implementation of Heron's square root algorithm instrumented with tagged error.

When we compile and run the code with tagged error, we get code listing 5.8.

```
sqrt:1.5000000000000000e+00 []
sqrt:1.41666666666667e+00 [add:75%, delta:24%]
sqrt:1.414215686274510e+00 [add:79%, delta:20%]
sqrt:1.414213562374690e+00 [delta:99%...]
sqrt:1.414213562373095e+00 [add:88%, delta:11%]
```

Code listing 5.8: Output for the implementation of Heron's square root algorithm instrumented with tagged error.

We observe that, perhaps counter intuitively, most of the numerical error in the output is produced during the addition step of the loop, not during the three operations used to compute delta. This proportion is not perfectly stable, as can be seen during the fourth operation, but it is due to the fact that the error is very small compared to the numbers (we have 15 significant digits) and thus easily overwritten in a single iteration.

Also note that during the first iteration, the error is exactly zero as 1.5 can be represented exactly with floating-point numbers. This explains why the initialization step has no contribution to the final numerical error.

It is our belief that, due to its flexibility, ease of use, and ability to target the sources of error that matter for the output, tagged error should be preferred to local, debugger-based, methods.

5.4 Tools

5.4.1 Numerical debugger

At first, to help with Shaman's usage, we included a numerical debugger, inspired by [Jézéquel and Chesneaux, 2008], and made possible by our fine-grained error representation. The user can use GDB [Stallman et al., 2002] or any classical debugger to pinpoint certain types of unstable operations such as cancellations and unstable comparisons.

The user can set a variety of compilation flags to indicate the kind of numerical errors that are of interest to him, such as unstable comparisons and cancellations. When one of those operations occurs, the program calls the *instability* function. The user can set a breakpoint on the function and it will pause the program, giving access to all variables and their numerical error.

While this lets the user quickly assess the localization of specific numerical errors, it might become overwhelming on large programs with thousands of cancellations and unstable tests, often due to only a small subset of operations. We resolved this problem by writing what we call a numerical profiler. It is a script that hooks itself onto GDB, records all breakpoint triggers, and produces a report with the line number, the operation name, and the number of occurrences (an example of output can be seen in section 5.3, code listing 5.6). This automation script gives the user an overall view of the numerical behaviors of their program.

It is important to note that those tools are now superseded by the tagged error which has the ability to detect sources of error that matter via a non-local analysis, avoiding the shortcomings mentioned in section 3.2.1.

5.4.2 Code instrumentation

A library-based implementation, such as ours, lets the user examine the instrumented code and compose our functions with their own to serve their specific purposes. This flexibility, however, requires access to the source code and the manual replacements of all floating-point types in a program.

Binary instrumentation tools, such as Valgrind [Nethercote and Seward, 2007] and Intel PIN [Luk et al., 2005], could make the instrumentation of a program easier but would obfuscate the instrumented code and are hard to compose.

We propose an automatic refactoring tool based on the Clang compiler [Lattner, 2008], which offers a good compromise between both approaches. It takes code, parses it with a state of the art C++ compiler (Clang), matches and replaces types and functions in the abstract syntax tree, and outputs the instrumented code. The refactoring tool is careful to produce warnings where a human operator should review the code, such as in the interfaces of *extern C* sections.

Part III

Evaluation and applications of the method

Table of Contents

Chapter 6

Accuracy

The first and foremost question when evaluating a new method to quantify the numerical error of a program is whether it is accurate enough to be useful. One can then wonder how it compares with the state of the art.

In the first section of this chapter, we compare the accuracy of Shaman and various state-of-the-art tools. We then evaluate Shaman on larger problems using different proxy to assess the numerical error that we should be measuring. Finally, we check whether the sources of error given by tagged error are reliable using targeted numerical improvements to compare the impact of various code sections.

6.1 Comparison with the state of the art

In the following, and in section 7.1, we compare Shaman with various tools. To do so, we picked at least one implementation for each of the most common approaches used to measure the numerical error. We choose these implementations because of their extensive usage in their category[1] and efficiency:

- MPFR [Fousse et al., 2007], with the MPFR C++ [Holoborodko, 2010] wrapper, which implements arbitrary precision arithmetic (tested with 100 and 200 bit of precision).
- Boost Interval [Brönnimann et al., 2006], which implements interval arithmetic.
- Verrou [Févotte and Lathuilière, 2016], which implements a form of stochastic arithmetic (statistical analyses were done with 80 samples).

[1] Thus, we do not include some alternatives, such as double-double [Dekker, 1971] (which we expect to be at least as precise as Shaman, due to the similarities in their design, but slower, due to the additional renormalization step) which, to the best of our knowledge, have never been used to measure numerical error (contrary to MPFR).

- Cadna [Jézéquel and Chesneaux, 2008], which implements a synchronous variant of stochastic arithmetic.

The following examples are used to demonstrate that Shaman works as expected and covers cases not covered by other methods. More realistic examples follow in the subsequent sections.

6.1.1 Rump equation

This polynomial was proposed by Siegfried M. Rump in [Rump, 1983]. When it is evaluated in double precision on $(\frac{1}{3}, \frac{2}{3})$, its output has 15 digits of precision, but when it is evaluated on $(10864, 18817)$ it returns 2 instead of 1, the output has no significant digits.

$$P(x, y) = 9 * x^4 - y^4 + 2 * y^2 \qquad \text{(Rump equation)}$$

It is an example of a computation that has completely different accuracy depending on its inputs. We consider it our baseline: a fairly simple case where all methods should perform well.

	P(10864,18817)	P($\frac{1}{3}$, $\frac{2}{3}$)
MPFR (100 bit)	1.0000000000000000e+00	8.02469135802469056e-01
MPFR (200 bit)	1.0000000000000000e+00	8.02469135802469056e-01
Boost Interval	[-1.400e+1, 2.000e+0]	[8.025e-1, 8.025e-1]
Verrou	μ=-1.0192e1 σ=6.8205e0	μ=0.8024e0 σ=9.9152e-17
Cadna	@.0	0.802469135802469E+000
Shaman	∼numerical-noise∼	8.02469135802469e-01
Double precision	2	8.02469135802469147e-01
Analytical result	1	$\frac{65}{81} \approx 8.02469135802469135e-01$

Table 6.1: Outputs of the tools for Rump's equation. We computed Verrou's metrics ourselves as it requires the user to collect and process the data manually over several runs of the instrumented program. The notations @.0 and ∼numerical-noise∼ denote a number with no significant digits according to their respective library.

Table 6.1 shows the raw outputs of each method. For ease of comparison, as the output format varies greatly from one method to another (single number, interval,

	P(10864,18817)	P($\frac{1}{3}$, $\frac{2}{3}$)
MPFR (100 bit)	0	15
MPFR (200 bit)	0	15
Boost Interval	≥ 0	≥ 15
Verrou	0	15
Cadna	0	15
Shaman	0	15
Expected results	0	15

Table 6.2: Estimation of the number of significant digits for Rump's equation.

statistics, number displayed with only its significant digits), all results will be converted in number of significant digits for the rest of this chapter.

The main point of this case study is not to discriminate between methods, as can be seen in Table 6.2 and as expected every method diagnose the outputs correctly, but to introduce the number of significant digits as a common metric to compare the accuracy of various algorithms and to illustrate the raw outputs of their associated implementations.

6.1.2 Trace of a parallel matrix product

One of the most common type of computations where the non-associativity of floating-point can be felt is parallel computations: if you compute a sum in parallel several times, the order in which intermediate sums are added is non-deterministic and hence, due to the non-associativity of IEEE-754 arithmetic, you might get different results.

In this example we initialize two double precision 1000 by 1000 matrices A and B with random values (using a fixed seed to insure reproducibility and fairness between experiments) and compute both $trace(A \times B) = \sum_{i,j} A_{ij}B_{ji}$ and $trace(B \times A) = \sum_{i,j} B_{ij}A_{ji}$ two times using a parallel reduction. Mathematics tell us that all these results should be equal but, due to the non-associativity of IEEE-754 arithmetic, we expect to get slightly different numbers giving us a way to confirm the number of significant digits given by our tools.

It is interesting to note that by printing only the significant digits, as can be seen in Table 6.3, Shaman reproduces the outputs of IEEE-754 arithmetic, but hides the impact of the numerical inaccuracies, and thus the non-associativity of

	Double precision	Shaman
trace(A*B) first run	2.4976998949002950e+15	2.4976998949003e+15
trace(A*B) second run	2.4976998949002900e+15	2.4976998949003e+15
trace(B*A) first run	2.4976998949002400e+15	2.49769989490029e+15
trace(B*A) second run	2.4976998949002350e+15	2.49769989490029e+15

Table 6.3: Outputs for the trace of a parallel matrix product. Each run is slightly different due to the non-associativity of floating-point arithmetic.

floating-point operations. Hence, while the outputs in double precision are different, the variability (due to a sum computed in parallel) disappears with Shaman.

	trace(A*B)	trace(B*A)
MPFR (100 bit)	14	15
MPFR (200 bit)	14	15
Boost Interval	$\geq$ **12**	$\geq$ **12**
Verrou	14	**14**
Cadna	14	**14**
Shaman	14	15
Expected results	14	15

Table 6.4: Estimation of the number of significant digits for the trace of a parallel matrix product.

A good property of this example is that we can make the problem as large as we want giving us a way to test tools on arbitrarily large number of operations (in this case more than two million operations). We see, in Table 6.4, that Boost Interval's results are pessimistic which is expected since interval arithmetic's bounds tend to grow larger with the number of operations. The other methods seem to stay robust despite a high number of operations.

6.1.3 A deterministic identity function

In this example we have a function (Identity) that should be the identity, but returns zero due to cancellations when computed in double precision:

$$id(x) = x \times \frac{(1 + 0.5^{100}) - 1}{0.5^{100}} \qquad \text{(Identity)}$$

This identity function is a simple cancellation inspired by [Panchekha et al., 2015] and requires more than 100 bit of accuracy for any problem to be detected as seen in Table 6.5. If we wanted to ensure that MPFR still returns an incorrect result with 200 bit of precision, one would just need to change the 0.5^{100} for a 0.5^{200} in the function definition.

	Double precision	Mpfr (100 bit)	Mpfr (200 bit)	Expected results
$id(4)$	0.	0.	4.	4.
$id(5)$	0.	0.	5.	5.
$id(5) - id(4)$	0.	0.	1.	1.
$id(5) - id(5)$	0.	0.	0.	0.

Table 6.5: Outputs for the deterministic identity computation. Only the results obtained with more than 100 bit of precision are accurate.

If we apply this function to two consecutive numbers and subtract the result, we expect to get one (Theoretical one). In practice, unless we have more than 100 bit of precision, it will return zero due to our identity function returning zero instead of its input: we need a system that is resilient to 100 bit cancellations to be able to detect problems in this function.

$$one(x) = id(x + 1) - id(x) \qquad \text{(Theoretical one)}$$

However, applying this function to the same number twice and subtracting the result should, and indeed does, produce a zero (Theoretical zero) independently of the errors introduced in our identity function since the computations, as incorrect as they are, are deterministic. To confirm that this function is exact, we would need a system that can deal with correlated errors properly.

$$zero(x) = id(x) - id(x) \qquad \text{(Theoretical zero)}$$

Table 6.6 shows that, as expected, one cannot identify the cancellation with MPFR and only 100 bit of precision. Furthermore, stochastic arithmetic (which

	$id(4)$	$id(5)$	$id(5) - id(4)$	$id(5) - id(5)$
MPFR (100 bit)	**17**	**17**	**17**	17
MPFR (200 bit)	0	0	0	17
Boost Interval	≥ 0	≥ 0	≥ 0	$\geq$ **0**
Verrou	0	0	0	**0**
Cadna	0	0	0	**0**
Shaman	0	0	0	17
Expected results	0	0	0	17

Table 6.6: Estimation of the number of significant digits for the deterministic identity computation.

is non-deterministic) and interval arithmetic (which cannot consider correlations between errors) are unable to tell us that $id(5) - id(5)$ is exact as they do not model the fact that both computations return the same, incorrect, result.

6.2 Validation of the accuracy

Now that we have established that Shaman behaves well on edge cases that cause problems to other methods, we want to validate the accuracy of Shaman using more realistic problems where accurate references are available. We first compare Shaman's error estimations with the estimations obtained using 10000 bit floating-point arithmetic on a LU factorization. Then, we compare Shaman's estimation with analytical values on an integration problem.

6.2.1 LU factorization

To demonstrate the accuracy of our algorithm, we applied it to the LU factorization of a double-precision 200 by 200 random matrix whose entries are uniformly sampled from $[-1, 1]$. The LU factorization algorithm decomposes a matrix into a lower triangular and an upper triangular matrix whose product is equal to the input matrix. It is a well-known algorithm that uses all arithmetic operators and is fundamental in linear algebra (used to solve linear systems, invert matrices, and compute determinants).

As the numerical stability of the algorithm depends on the magnitude of its pivots, most implementations use the partial pivoting strategy. The test is

performed twice, with and without partial pivoting. The condition number of our test matrix, defined as the ratio of its largest and smallest eigenvalues, is roughly 700. Thus, one would expect the numerical error to be sensibly smaller when using partial pivoting.

To evaluate the accuracy of Shaman's estimation of the error, we also perform the computation using MPFR and 10000 bit of precision. While using 10000 bit of precision is too slow and memory intensive to be used on larger problems, it should be accurate enough to be used as a reference. Note that we set aside any cell that has an infinite number of significant digits according to MPFR (meaning that MPFR and double precision reached the same number) when computing average numbers of significant digits.

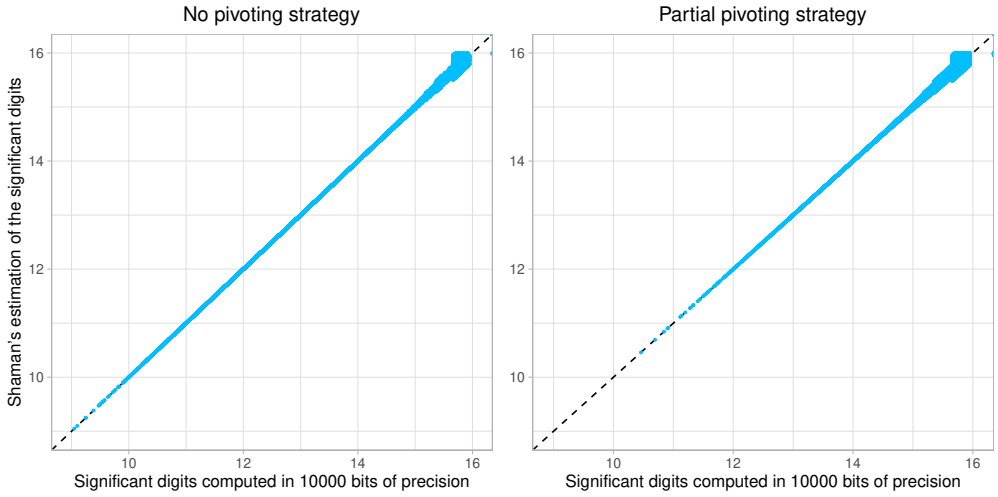


Figure 6.1: Estimation of the number of significant digits for the LU factorization algorithm computed in double precision. Each dot corresponds to a cell in the non-zero half of either the L or U matrix produced by the decomposition. The dashed line represents the equality between both estimations of the number of significant digits.

Without any pivoting strategy, we measured an average of 13.03 significant digits and a mean absolute difference between our estimation and the 10000 bit arithmetic estimation of 0.004 significant digits. With a partial pivoting strategy, we measured an average of 14.80 significant digits, confirming that the algorithm is more numerically stable. We also measured a mean absolute difference between our estimation and the 10000 bit arithmetic estimation of 0.028 significant digits. In both cases, the difference between our estimation and the 10000 bit arithmetic estimation of the numerical error is close to the machine epsilon (roughly 10^{-16} for 64 bit double precision).

Looking at Figure 6.1, one can observe that Shaman's estimation of the number of significant digits is consistent with the reference. It tends to be noisier when the number of significant digits is large which makes sense as it means that the numerical error is close to machine epsilon, making it harder to manipulate accurately. However, its relative precision increases sensibly with the magnitude of the numerical error, meaning that Shaman gets more accurate in its estimation as the number of significant digits of the output of this computation decreases. A good behavior as the difference between 15 and 16 significant digits is less likely to matter compared to the difference between 5 and 6 significant digits when one wants to evaluate if a program is precise enough for a given use-case.

6.2.2 Integration by the rectangle rule

We also evaluated the accuracy of Shaman on the integration of the cosine function between 0 and $\frac{\pi}{2}$ using the rectangle method (this test case comes from the authors of Verrou [Févotte and Lathuilière, 2016] and, in particular, [Févotte and Lathuilière, 2017]).

In infinite precision the only source of error is the discretization error of the integration which reduces in $O(\frac{1}{n})$ as the number of rectangles increases (meaning that the step size gets finer) until the result reaches 1, the known analytical value of the integral. In finite precision, here *float* precision (32 bit), there are two sources of error: the discretization error and the numerical error. This is particularly interesting because, as the discretization error reaches zero, it gives us a reliable estimate of the numerical error.

When we plot the difference between the result of the integration and the analytical value as a function of the number of rectangles (Figure 6.2), we observe that while it first decreases (which is predicted by the decrease in discretization error), it then starts to become noisier and increases.

This behavior is explained when we look at Shaman's estimation of the numerical error (here we display the absolute value of the raw numerical error and not the number of significant digits). It can be seen that the numerical error increases with the number of rectangles and ends up becoming the dominant source of error[2], the estimation of the numerical error perfectly overlapping with the error computed analytically.

[2] A similar observation is at the root of Jean Vignes's "test d'arrêt optimal" [Vignes, 1984], a stopping criteria for iterative algorithms.

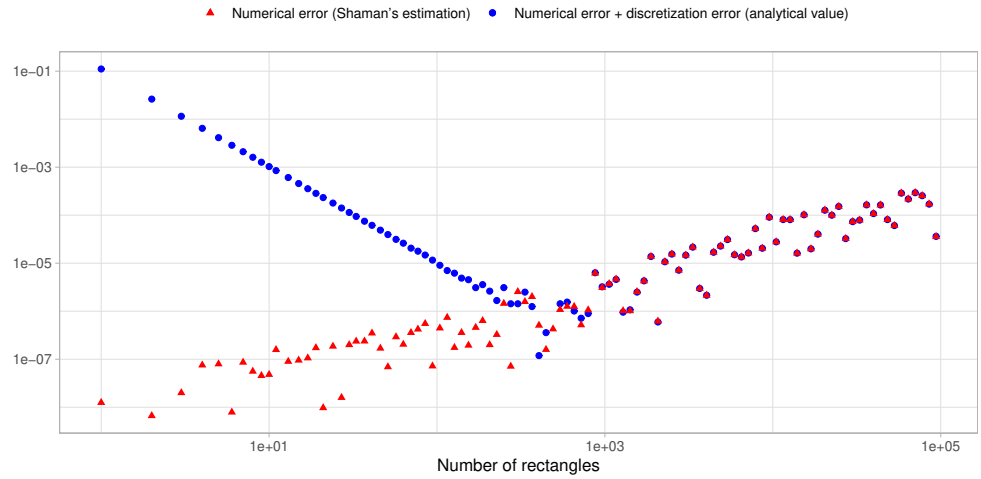


Figure 6.2: Absolute value of the error as a function of the number of rectangles used for the integration of the cosine function between 0 and $\frac{\pi}{2}$ using the rectangle method. Both axes are displayed in a logarithmic scale.

6.3 Tagged error

Evaluating the accuracy of the tagged error algorithm is trickier as there is no algorithm for the localization of sources of error that is similar enough to be used as ground truth or point of comparison. What we can do is reduce the numerical error locally, replacing parts of an algorithm with numerically more stable or compensated alternatives, and check whether tagged error was able to predict the impact of the modification.

6.3.1 The conjugate gradient algorithm

In this section we will study the conjugate gradient algorithm, an iterative algorithm that solves linear systems under the hypothesis that they are symmetric and positive-definite. It is commonly used due to its fast convergence.

As can be seen in algorithm 14, the conjugate gradient algorithm can be split into five sections, an initialization phase (*initialization*) and a loop composed of a matrix-vector product (*matrix-vector product*), two dot products (*dot product 1* and *dot product 2*), and various smaller operations (*loop body*).

Algorithm 14 ConjugateGradient(A, b, x)

$residualVector \leftarrow b - A * x$ ▷ **initialization**
$searchDirection \leftarrow residualVector$
$squaredResidual \leftarrow residualVector \cdot residualVector$
while not $convergenceCriteria(squaredResidual)$ **do** ▷ **loop body**
  $Ad \leftarrow A * searchDirection$ ▷ **matrix-vector product**
  $stepSize \leftarrow squaredResidual/(searchDirection \cdot Ad)$ ▷ **dot product 1**
  $x \leftarrow x + stepSize * searchDirection$
  $residualVector \leftarrow residualVector - stepSize * Ad$
  $newSquaredResidual \leftarrow residualVector \cdot residualVector$ ▷ **dot product 2**
  $ratio \leftarrow newSquaredResidual/squaredResidual$
  $searchDirection \leftarrow searchDirection * ratio + residualVector$
  $squaredResidual \leftarrow newSquaredResidual$
return x

Our tests are made with a random input vector, a random matrix of size 1000 and a random right-hand side of size 1000. The input vector and right-hand side are reused from one experiment to the other. The matrix is generated to have a given condition number[3], defined as the ratio of the absolute values of its smallest and largest eigenvalues. The algorithm stops when it either reaches 1000 iterations or a residual of 10^{-10}. We measure the mean of the absolute value of the numerical on the output vector to have a single number representing the full vector.

6.3.2 Analysis with tagged error

Figure 6.3 shows the evolution of the numerical error as a function of the matrix condition number, as measured by Shaman.

Following the code splits suggested in 6.3.1, we instrumented the code with tagged error. Figure 6.4 shows the raw value of the numerical error associated with each code section. As the log scale can be misleading, we also display the error associated with each term as a percent of the total error in Figure 6.5. The main observation is that the matrix-vector product quickly becomes the cause for the vast majority of the numerical error in the output.

A matrix of condition number 10^{26} exhibits a typical behavior: namely, the output has a mean relative error of 5.4 times the value of the output (meaning

[3]In practice, we use a two-step process to generate the matrix. We first sample values uniformly and shift them to produce a diagonal matrix with the target condition number. We then generate a random orthogonal matrix and multiply the diagonal matrix on both side (by the orthogonal matrix and its transpose) to produce a symmetric positive definite matrix with the desired condition number.

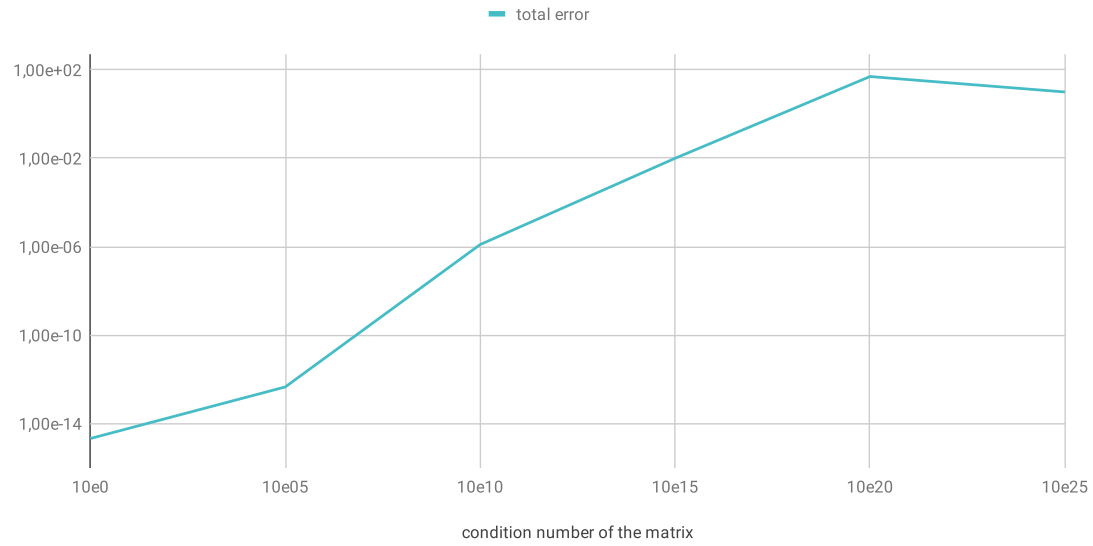


Figure 6.3: Evolution of the numerical error in the output of the conjugate gradient algorithm as a function of the matrix condition number. Note that the numerical error is displayed in log scale.

that it is mostly numerical noise), ninety-nine percent of which comes from the matrix-vector product, and an absolute residual of about 14. The algorithms did not converge and stopped only because it reached the maximum number of iterations allowed.

6.3.3 Introducing one compensated operation

We then replaced one operation, a dot product or the matrix product, by an equivalent compensated algorithm to reduce their contribution to the numerical error sensibly. Tagged error tells us that the vast majority of the numerical error comes from the matrix-vector product and that replacing it should be much more impactful.

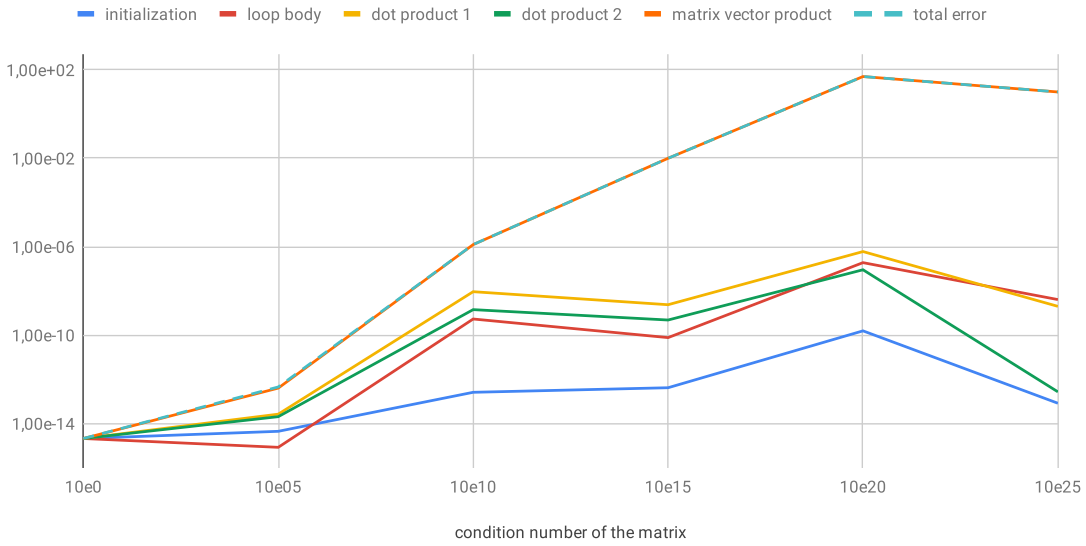


Figure 6.4: Distribution of the numerical error in the output of the conjugate gradient algorithm as a function of the matrix condition number. Note that the numerical error is displayed in log scale.

Compensated operation	Numerical error	Residual
None	5.4	14
Dot product 1	0.56	6.9
Dot product 2	60.5	$6.8e5$
Matrix vector product	$1.5e^{-}7$	3.5

Table 6.7: Relative numerical error, computed on the mean of the output vector, and absolute residual associated with the solution of the linear system when an operation is replaced by a compensated algorithm.

Table 6.7 illustrates the residual and relative error observed once one compensates a single operation. As predicted, compensating the matrix-vector product has a major impact while the dot products give much lower benefits (compensating dot product 2 is even detrimental to the algorithm!).

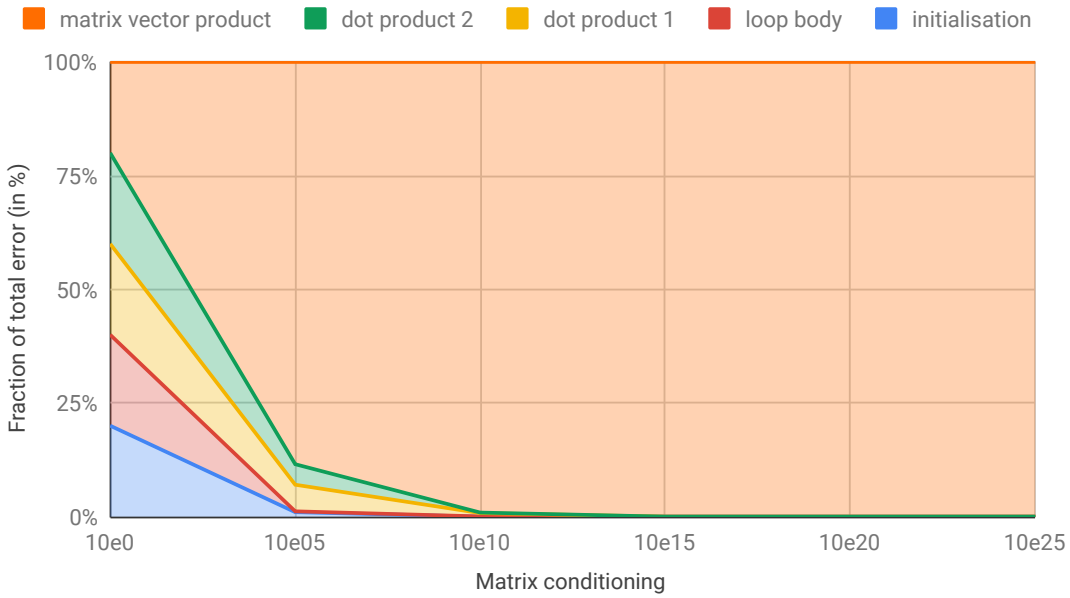


Figure 6.5: Distribution of the numerical error in the output of the conjugate gradient algorithm, in percents of the total error.

6.3.4 Introducing two compensated operations

As expected, once we compensate the matrix vector product, the product stops producing ninety-nine percent of the numerical error, leading to the following repartition:

$$[dotproduct1 : 51\%, loopbody : -21\%, dotproduct2 : 17\%, initialization : 7\% \dots]$$

Dot product 1 causes about half of the numerical error, much more than dot product 2, despite both algorithms doing the exact same number of arithmetic operations.

Compensated operation	Numerical error	Residual
Matrix vector product only	$1.5e^{-}7$	3.5
Matrix vector product and Dot product 1	$9.4e^{-}8$	3.3
Matrix vector product and Dot product 2	$3.0e^{-}7$	3.5

Table 6.8: Relative numerical error, computed on the mean of the output vector, and absolute residual associated with the solution of the linear system when the matrix vector product and an additional operations are replaced by compensated algorithms.

Table 6.8 illustrates the residual and relative error obtained by compensating both the matrix-vector product and a dot product. It shows that compensating dot product 1 reduces the numerical error by a further 37 percent, while compensating dot product 2 was, once more, detrimental.

While the reduction in numerical error is not as large as predicted (37% instead of 51%), it can be explained by the dynamic nature of the algorithm which, once the second compensated algorithm was introduced, took fewer iterations to converge and thus behaved slightly differently. Furthermore, the numerical error of a computation stems from an interdependent system, if a section improves its numerical stability significantly, then it will produce different outputs and thus the other sections will behave differently.

Once dot product 1 has been compensated, it is interesting to observe that the main sources of error becomes the various operations in the main loop and still not the second dot product:

$$[loopbody : 73\%, initialization : 11\%, dotproduct2 : 11\% \dots]$$

It is our hope that this short study adequately demonstrates the power of tagged error as a means to identify the sources of numerical error and explore ways to make a computation more numerically stable.

As a side note, algorithms with targeted compensated operations might be of interest in linear algebra as a way, orthogonal to preconditioners, to deal with badly conditioned problems or reduced precision algorithms. It might be especially interesting when dealing with sparse problems, where the additional computation and increased arithmetic intensity, due to the introduction of the compensated operations, might have a lower impact on the computing time.

Chapter 7

Cost of measuring numerical error

One of our goals was to develop a method fast enough to be viable for the analysis of large numerical programs. In this chapter we evaluate the runtime overhead introduced when instrumenting a code with the Shaman library. We first compare Shaman with the state-of-the-art on a variety of benchmarks before evaluating the overhead of tagged error as a function of the number of tags and, finally, gathering overhead data on all the programs that have been studied in this dissertation.

7.1 Comparison with the state of the art

In this section we compare Shaman with the tools introduced in Section 6.1 using highly optimized benchmark codes as well as a more realistic numerical computation. Please note that the computing times given for Verrou and Mpfr are lower bounds as, for Verrou, we report computing time for only one run while, stochastic arithmetic requires several runs in order to draw any conclusion. One would need to multiply Verrou's computing time by a factor of five or more to take this fact into account. Along the same line, we do not take into account the fact that, to draw conclusion on the numerical error when using Mpfr or any higher precision arithmetic implementation, one would need to compare a higher precision run with a double precision run.

We instrumented four programs. The first three are the tasks that deal with floating-point arithmetic in the computer benchmark game [Gouy, 2020], a well-known benchmarking suite that is used to compare the peak performances of programming languages on different tasks. The last program of our selection is the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics code (Lulesh 1.0 [Karlin, 2012]), a proxy program for performance benchmark for exascale computing. To sum up, our experiments consist of instrumenting and analyzing:

- n-body: N-body simulation.

- Spectral norm: computing an eigenvalue using the iterated power method.
- Mandelbrot set: generating the Mandelbrot set at a given resolution; it is the only parallel benchmark of the set.
- Lulesh 1.0: solving explicit hydrodynamics equations on a collection of volumetric elements.

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

We also instrumented the serial version, as some tools would not lend themselves to the parallel versions, of Lulesh 1.0 on a grid of size 10 in order to compare different tools on a program that is representative of the kind of computations that are actually done in high-performance computing. We expect to observe a very representative overhead on this benchmark.

Each program was compiled with GCC 7.3.0 and the -O3 optimization flag and ran on a 4-cores Intel(R) Xeon(R) CPU E3-1220 v3 @ 3.10GHz with 16 GB of RAM. Measures presented below correspond to the minimum computing time over thirty runs (while the average running time produces a similar plot, it is less appropriate to estimate the intrinsic running time independent of the perturbations as perturbations can only increase the running time). Since the variation coefficient was always below 2%, for the sake of readability, we do not include error-bars.

As shown in Figure 7.1, Shaman displays the lowest overhead on every program. Moreover, it is interesting to observe that, while it takes eight operations to compute an addition with our formula, the increase in run time using Shaman stays well below a factor eight for most programs.

One explanation is that the computations spend a non-negligible time on non-numerical operations, but it seems unlikely given that those are numerically intensive programs. An alternative and more likely explanation is that, since the compiler has full access to the instrumented code (which would not be the case if we were instrumenting at the binary level) and since there are no tests inside our arithmetic operators, the compiler can still vectorize and use instruction-level parallelism. Furthermore, since our representation takes only twice as much memory as the original floating-point representation, it increases the arithmetic intensity (see section 7.2) which is beneficial on modern processors [Yang et al., 2018].

We note that the increase in run time on the Mandelbrot set computation is significant for all tools, especially for Mpfr. This might be due to the fact that it

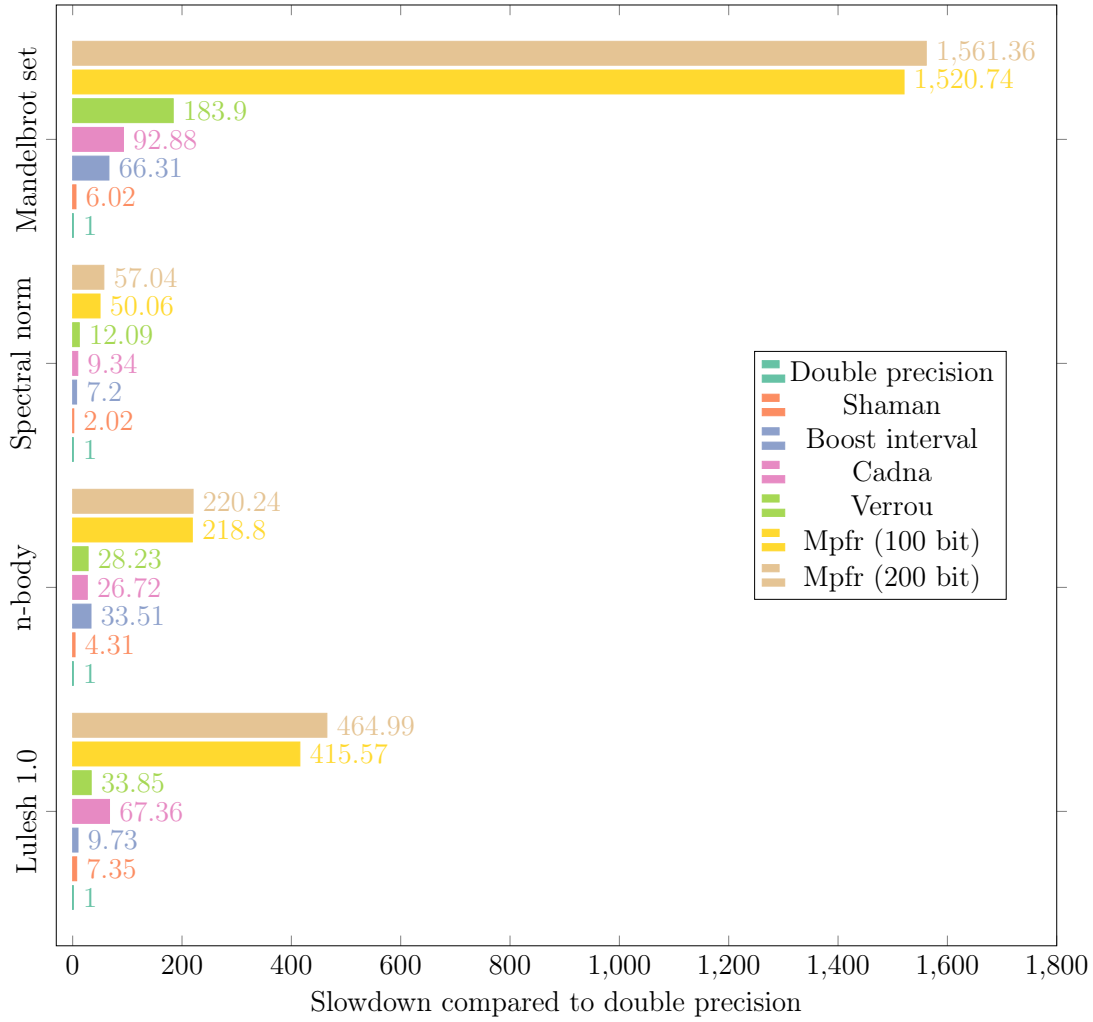


Figure 7.1: Overhead of the Shaman library compared to the state of the art.

is the only parallel program of our benchmark or that its operations are highly optimized making its performance more sensible to source code modifications.

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

7.2 Arithmetic intensity

Arithmetic intensity is defined as the number of floating point operation per byte of memory accessed. A program is said to be *memory bound* if is has a low arithmetic intensity, meaning that the processor spends time waiting while data is fetched from memory. On the opposite side of the spectrum, a program is said to be *compute bound* if it has a high memory intensity, the processor is fully utilized and can be feed as soon as data is requested [Marques et al., 2017].

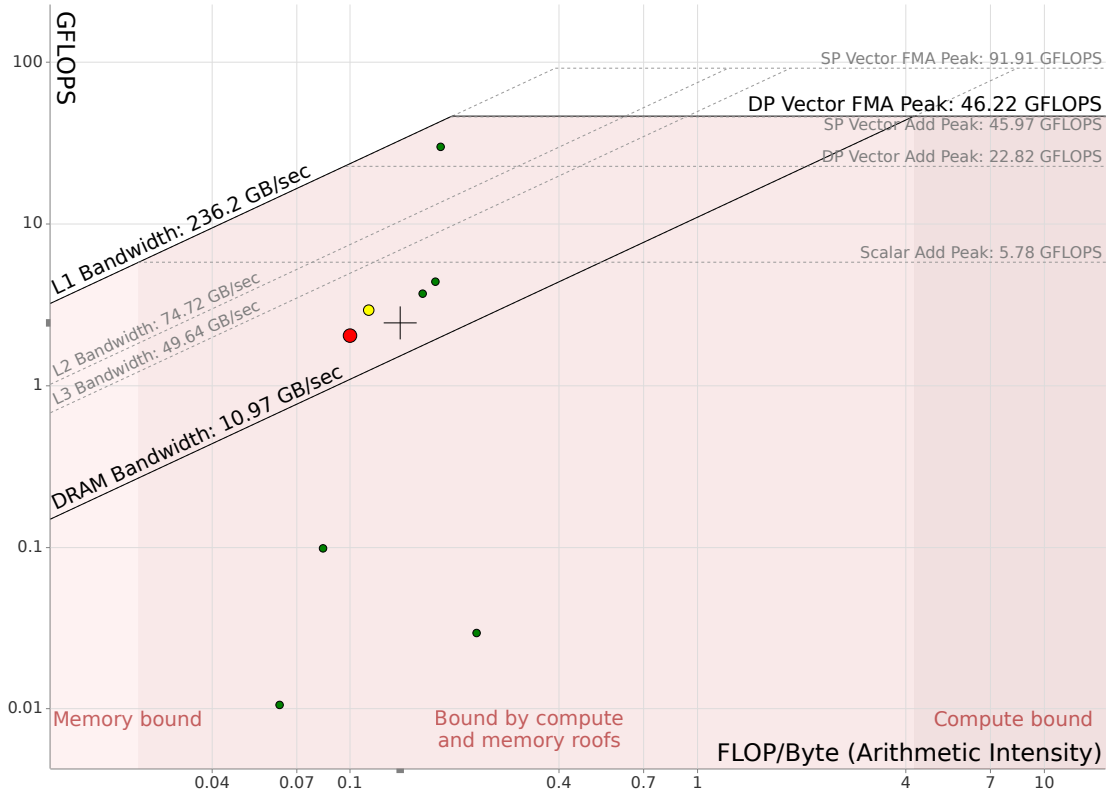
	Shaman's arithmetic operators			Heron's algorithm	
Operator	Addition	Multiplication	Division	IEEE-754	Shaman
$\pm$	9	2	3	11	141
$\times$	0	3	1	6	28
$/$	0	0	2	10	20
fma	0	1	1	0	16

Table 7.1: Number of arithmetic operations used in Shaman's algorithms and an execution of either the instrumented or non-intrumented version of Heron's algorithm.

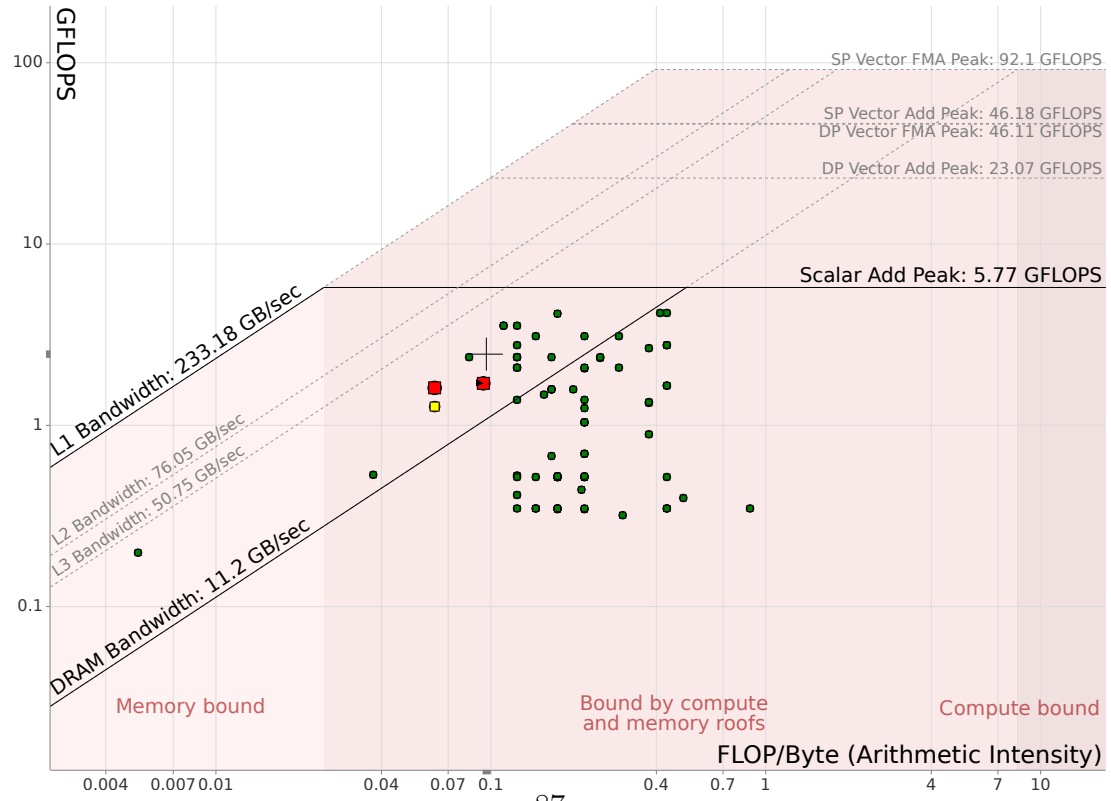
Table 7.1 illustrates the number of operation ($+, -, \times, /$ and *fma*) used to implement Shaman's arithmetic operators and called during both the instrumented and non-instrumented version of Heron's algorithm (as implemented in section 5.3.1). Encapsulated error uses twice as much memory as floating point arithmetic but two to ten times as many arithmetic operations, most of them additions and some of them *fused-multiply-add*, a processor instruction that is often idle during dense arithmetic computations. Thus, we can expect an increased arithmetic intensity, meaning that part of encapsulated error's overhead should be amortized by an increased processor usage as long as the program is not compute bound.

Figure 7.2 shows the arithmetic intensity for both IEEE-754 double precision and Shaman Sdouble precision on Lulesh 1.0 as a roofline plot. One can see that once Shaman is used, the measures shift up and to the right meaning that both the minimum number of operation per seconds and the arithmetic intensity increased. Furthermore, the program goes out of the fully memory bound section (the slanted zone on the left of the roofline model) to becomes both memory and compute bound.

This confirms our hypothesis that Shaman makes a more efficient use of the processor which, at least partly, explains why its overhead is not as important as would be predicted by the raw number of arithmetic operations.



(a) IEEE-754 double precision.



(b) Shaman Sdouble.

Figure 7.2: Roofline plot comparing the arithmetic intensity of Shaman and double arithmetic on Lulesh 1.0. The x-axis represents the arithmetic intensity while the y-axis is the number of operation per seconds. Green, yellow and red points represent measures, randomly sampled, colored as a function of their associated computing time. Plot produced with Intel advisor [Marques et al., 2017].

7.3 Tagged error

Tagged error's running time is a direct function of the number of tags. Therefore, we decided to measure its runtime overhead, compared to a non-instrumented execution, as a function of the number of tags. This should be an affine function as, in our implementation, the time to do a single floating-point operation instrumented with tagged error is proportional to the number of tag.

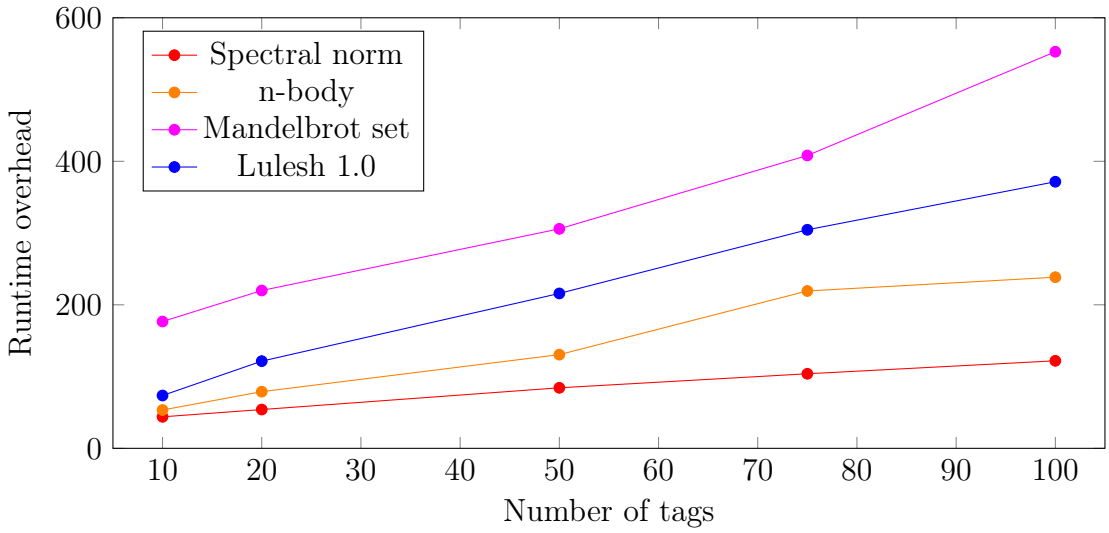


Figure 7.3: Runtime overhead of the tagged error algorithm, compared to double precision, as a function of the number of tags.

Figure 7.3 illustrates the computing time overhead due to the addition of tagged error as a function of the number of tags. As predicted, it is an affine function. Fitting a linear model on the data, we observe a slope going from 1 to 4 depending on the computation and a value at the origin of about 35 except for the Mandelbrot computation which starts at 130.

It is interesting to note that the Mandelbrot computation is, once more, the most impacted. Its runtime overhead seems to be roughly 4 times larger than expected which might be linked to the fact that it uses 4 cores.

One can get a feel for the number of floating-point numbers used for each program by looking at the memory overhead (which should also be an affine function of the number of tags).

The memory overhead, which is displayed in Figure 7.4, shows affine functions with a slope going from $\frac{1}{1000}$ to $\frac{1}{3}$ and a value at the origin of about one which is very stable from one program to the other. Lulesh is the program with the highest

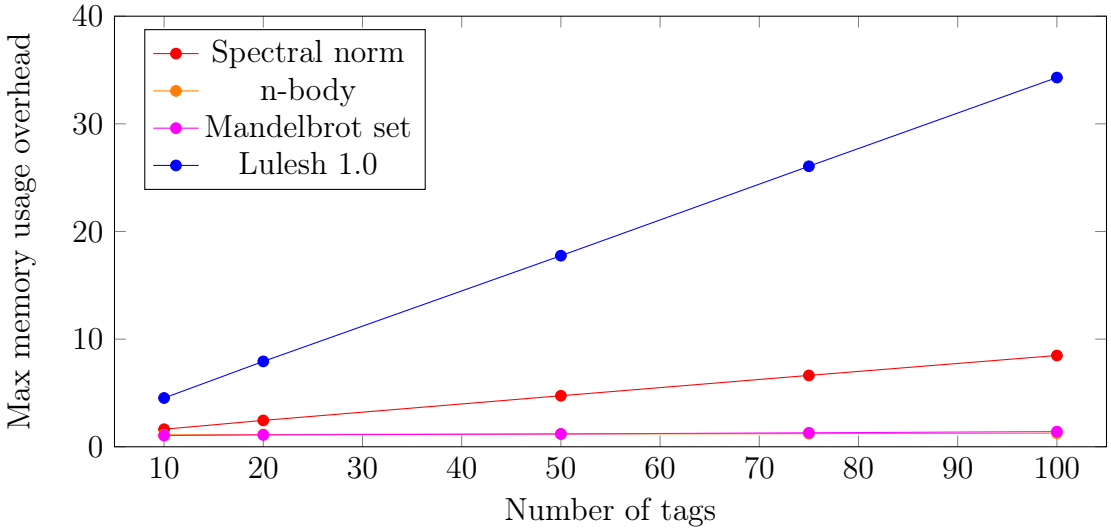


Figure 7.4: Memory overhead of the tagged error algorithm, compared to double precision, as a function of the number of tags. The Mandelbrot set and n-body plots are overlapping and stay very close to 1.

slope which is a direct consequence of the fact that it has the largest quantity of floating-point numbers used at a given time.

The plots are notably less noisy, better fitted by a linear model, than the one obtained for the computing time overhead. Probably because they are less impacted by interactions between components of the program.

The low slopes mean that the memory overhead is negligible at first and grows very slowly with the number of tags (you need at least 6 tags to double your memory usage), a good property as the memory usage of large simulations could, otherwise, quickly get unwieldy when using dozens of tags.

It seems important to indicate that, in practice when analyzing a given program, we rarely needed more than 10 tags as the analysis was usually done in three or four runs of increasing granularity.

7.4 Exhaustive overhead analysis

In this section we collect the overhead observed while using Shaman on *all* the programs, and even smaller algorithms, in this thesis. This is meant as a way to get an overview of the slowdown one can expect while using Shaman.

7.4.1 Encapsulated error

Figure 7.5 synthesizes the overhead observed for all programs featured in the thesis, we only omitted the computing times of Felix 1.0 and 2.0 as it was instrumented with a much older version of Shaman. Here, we are interested in the range of behaviors observed when instrumenting a program with Shaman rather than the behavior of individual programs. The main conclusion made from this figure is that the overhead can go anywhere from a slight decrease in run time (that was only observed on very fast, toy, test cases and might be due to measurement noise) to a factor of about 9 (on a program that spends the vast majority of its time doing dense linear algebra). Overall, an overhead of about a factor 7 is a realistic expectancy when instrumenting a code with Shaman.

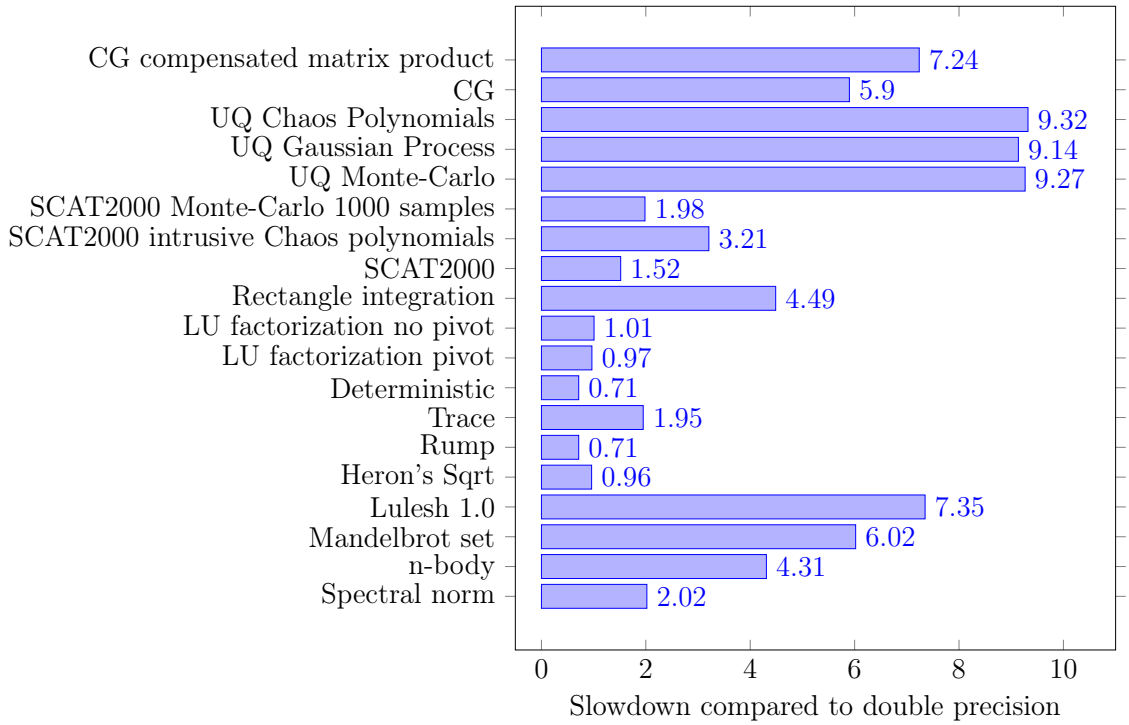


Figure 7.5: Overhead of the Shaman library compared to double precision.

Figure 7.6 displays the computing time of the non-instrumented and instrumented version of all the programs to give a sense of their respective scale.

It is interesting to note that the overhead does not increase monotonously with the computing time. While the fastest programs are the only ones to exhibit an overhead below one (probably because it is lost in measurement noise), the overhead alone cannot be used to discriminate between slow running programs and faster ones.

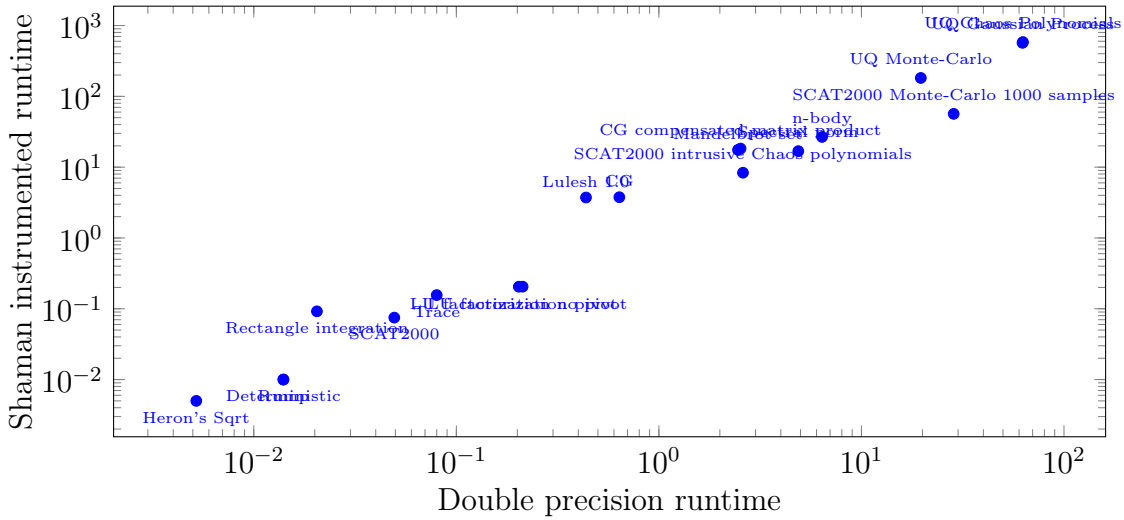


Figure 7.6: Comparison of computing times between double precision and Shaman instrumented code in log-log scale.

This is well illustrated by the SCAT2000 and UQ programs. While these programs have similar computing times (in particular the Monte Carlo versions), their overhead are drastically different (below 4 and around 9 respectively), probably due to the UQ program relying on heavily optimized linear algebra kernels that are more impacted by the instrumentation. Meanwhile, despite the various versions of these programs having very different computing time, they have similar overheads.

7.4.2 Tagged error

Figures 7.7 gives the overhead of the introduction of tagged error for the programs covered in the previous section. We excluded SCAT2000 from our instrumentation as, for historical reasons, some of its components write data on file to read it later which leads to problems as its parser cannot process the tagged error output format (a problem that does not occur when using Shaman without tagged error as its output is displayed as a normal number, with no tag related information, which can be parsed normally).

We used 10 tags as it is representative of the number of tags we usually used to search for the sources of numerical error in a program.

Looking at the plot, and ignoring programs too short to be impacted by the introduction of tagged error (all programs with an overhead below a factor of two can be put in this category), it is clear that tagged error has a large overhead, somewhere between a factor of 40 and 100 for most programs (once more, the

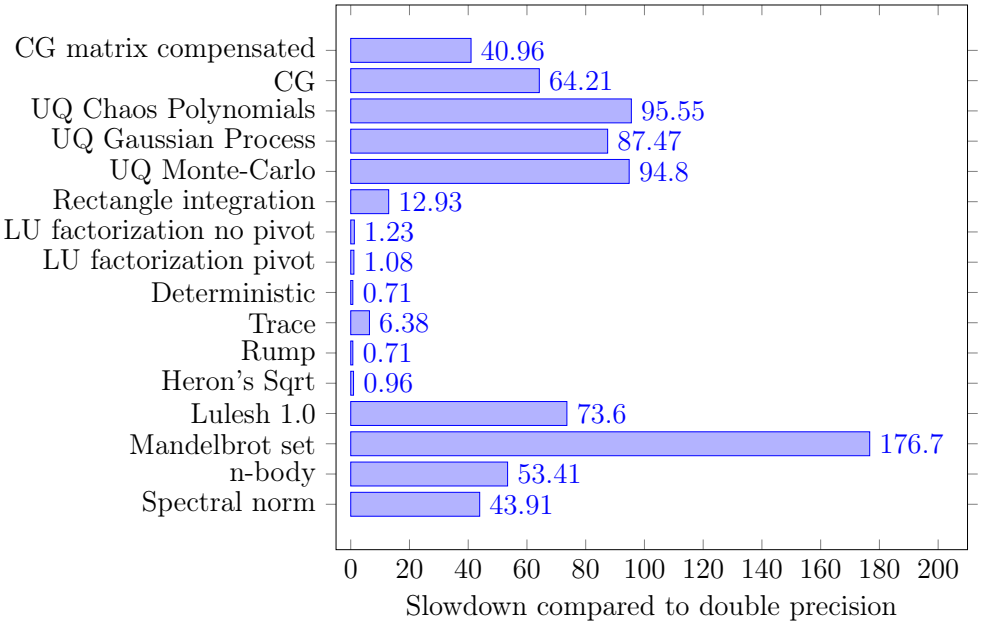


Figure 7.7: Runtime overhead of the tagged error algorithm, using 10 tags, compared to double precision.

computation of the Mandelbrot set is a notable outlier). With ten tags, an overhead of about 70 is a realistic expectation: about the number of tags time Shaman's overhead.

This large overhead, however, is not as problematic as it would be with Shaman as tagged error focuses on extracting very rich information from a single run and is expected to be only used on use cases where a significant numerical error has been spotted with Shaman.

7.5 Conclusion

Instrumenting a program with Shaman makes it about seven times slower but the overhead appears to be highly computation dependent, going from no perceptible overhead to making a program nine times slower. While this might seem like a large value, it is highly competitive with the current state-of-the-art (see section 7.1) making it particularly suitable for the analysis of long-running programs common in simulation and high performance computing. Tagged error, while more powerful, is notably more costly. One can expect that using ten tags will result in making a program seventy times slower. Furthermore, its cost grows linearly with the number of tags.

The wide range of overheads observed might be linked to the arithmetic in-

tensities of the programs instrumented. As the increase in number of operations outweighs the increase in memory use, programs that are at peak performance (a maximal number of operations per bytes of memory used) see a slowdown proportional to the increase in arithmetic operations. However, programs that spend time waiting while data is fetched from memory to the processor are much less impacted since they can use this waiting time to do the additional computations. Paradoxically, the instrumentation gets the program closer to peak performance by increasing its arithmetic intensity.

Chapter 8

Applications to physical simulation

In this chapter we synthesize results obtained by applying Shaman to actual programs from the field of physical simulation.

We present three programs. First, Felix [Regnier et al., 2016, Regnier et al., 2017], a code used for fission simulation which has been used to check whether our tools could scale to a non-trivial code base. Second, SCAT2000 [Bersillon, 1988], a code from the world of nuclear simulation which we analyze to determine whether its outputs where reliable. Third, we study the numerical error of a full pipeline, including a physical model and various metamodels used to compute its uncertainty, in order to check whether the numerical error impacts uncertainty quantification.

8.1 Instrumentation of a large fission simulation

Felix [Regnier et al., 2016, Regnier et al., 2017], a code used for fission simulation, has been the first large program that we instrumented with Shaman. Our goal with this study was not to validate particular outputs, but rather to check whether our tools could scale to a non-trivial problem.

In this section we quickly introduce Felix before detailing how our instrumentation of both Felix 1.0 and Felix 2.0 went. We conclude with the impact of this test case on the development of our tools.

8.1.1 Problem statement

Felix is used to solve the collective Schrödinger equation in a finite element basis with the time dependent generator coordinate method (TDGCM) in N-dimensions under the Gaussian Overlap Approximation (GOA). It is given in Equation 8.1 where i is the imaginary unit, $\hbar$ the reduced Planck constant, t the time, q a point

in space, g is the function describing the dynamic of the system (linked to the wave function), B is the collective inertia tensor, and V is the potential.

$$i\hbar\frac{\partial}{\partial t}g(q,t) = \left[-\frac{\hbar^2}{2}\sum_{k,l}\frac{\partial}{\partial q_k}B_{kl}(q)\frac{\partial}{\partial q_l} + V(q)\right]g(q,t) \tag{8.1}$$

There are two versions of the code, Felix 1.0 [Regnier et al., 2016], the first version for which the authors rolled their own linear solvers and Felix 2.0 [Regnier et al., 2017] which uses the Eigen linear algebra library [Guennebaud et al., 2010] to solve its linear systems and introduces additional functionalities. In both cases, Felix has two main components. First, it discretizes the N-dimensional collective space using the Galerkin finite element method on a basis of Laplace polynomials. Those polynomials are integrated analytically in Felix 1.0 (meaning that numerical error can only come from the polynomial coefficients computation and not from the integration) and with an adapted quadrature in Felix 2.0. Second, it solves for the time evolution using either the Crank-Nicholson scheme in Felix 1.0 (which requires an iterative QMR algorithm with a Jacobi preconditioner) or a Krylov method in Felix 2.0 (using the Arnoldi iterative algorithm).

Both Felix 1.0 and Felix 2.0 have about 10000 lines of code spread on a hundred files and use OpenMP to run in parallel. These were selected as our first large test programs as they do not depend on Fortran subroutines and because both come with input values to model the following four test cases, some of which have been validated with analytical solutions:

- the oscillations in a 1D harmonic potential, for which an analytical solution is available,
- the oscillations in a 2D isotropic harmonic potential, for which an analytical solution is available,
- the propagation of a free wave packet, a very simple case which was used to check that the errors introduced in the computation were low enough,
- the induced fission on a 239 plutonium target, a much larger problem.

The code takes a set of points, measurements at those points, a description of the geometry of the problem, and an initial wave function as input. It outputs a series of wave functions, which we will call gFunctions following Felix's terminology, sampled at regular intervals and the average energy of the solution as a function of the number of iterations.

We decided to instrument both versions of Felix and see if we could run those test cases and detect meaningful differences between Felix 1.0 and Felix 2.0.

8.1.2 Instrumentation and numerical stability

Felix 1.0

Even before the analysis started, due to its size (about hundred files), Felix 1.0 pushed us to develop an automatic instrumentation tool: at first a python script and now, as detailed in Section 5.4.2, a proper Clang based refactoring tool.

Once instrumented, Shaman indicated that the outputs of two of the test cases, the oscillations in a 1D harmonic potential and the induced fission on a 239 plutonium target, had severe numerical inaccuracies (the average energy had a handful of significant digits, but the gFunction were very inaccurate). Using the numerical debugger on the simulation of the oscillations in a 1D harmonic potential to pinpoint their source quickly became impractical as there were thousands of cancellations (as can be seen in code listing 8.1), most of them benign, which made a breakpoint type of approach unscalable.

```
*** SHAMAN ***
There are 179750963 numerical instabilities
40552 UNSTABLE CANCELLATION(S)
116903 UNSTABLE DIVISION(S)
179500506 UNSTABLE MULTIPLICATION(S)
29974 UNSTABLE MATHEMATICAL FUNCTION(S)
0 UNSTABLE POWER FUNCTION(S)
63028 UNSTABLE BRANCHING(S)
```

Code listing 8.1: Output of the numerical debuggeur for Felix 1.0.

This led to the development of the numerical profiler, a program able to catch all breakpoints and print a summary of the results, as presented in Section 5.4.1. Using the numerical profiler, we were able to assess that most cancellation occurred during the computation of the Laplace polynomial coefficients. However, rewriting the corresponding code with a numerically stable algorithm (using a compensated dot product) did not yield improvements in Shaman's estimation of the output numerical error, meaning that it is unlikely to be the cause of the inaccuracies. This highlighted the limitations of fully local methods to find the sources of numerical error and later caused the development of tagged error as a means to evaluate the numerical error caused by a section of the code in a non-local way.

The simulation of the oscillations in a 1D harmonic potential, which was flagged as inaccurate, had been previously validated with an analytical solution which lead to doubts on the accuracy of Shaman at that scale. We thus decided to analyze the code with an orthogonal method, stochastic arithmetic, using Verrou which had already been used on large test cases by its authors. Verrou's results (obtained by computing the standard deviation of each output value across a hundred runs

of the program with stochastic arithmetic enabled) corroborated those obtained with Shaman, namely that the gFunctions outputted for the oscillations in a 1D harmonic potential where very inaccurate. While validating Shaman's outputs, this did not explain how both methods could be so pessimistic on a computation that was analytically validated. This was later explained by the fact that the analytical solution was not used to validate this test case (contrary to the other analytically solvable test cases that are numerically stable according to Shaman). Instead, it was validated with a proxy value (the deviation of the modulus of the numerical solution [Regnier et al., 2016]), which checks a property of the solution but does not actually compare it with an analytically predicted value.

Felix 2.0

Felix 2.0 is the current version of the code and is still used by physicists to solve the collective Schrödinger equation. It has the same test cases as Felix 1.0, but an improved code base with both the polynomial initialization and the linear algebra fully rewritten.

Once instrumented, Shaman indicated that all the previously inaccurate test cases were now accurate, all outputs had between 7 and 15 significant digits, while Felix 2.0 only displays the first 6 digits of its outputs. Displaying the number of cancellations for the simulation of the oscillations in a 1D harmonic potential, we obtained Code listing 8.2.

```
*** SHAMAN ***
There are 99438313 numerical instabilities
2482088 CANCELLATION(S)
1959750 UNSTABLE DIVISION(S)
90347107 UNSTABLE MULTIPLICATION(S)
196591 UNSTABLE MATHEMATICAL FUNCTION(S)
0 UNSTABLE POWER FUNCTION(S)
4452777 UNSTABLE BRANCHING(S)
```

Code listing 8.2: Output of the numerical debugger for Felix 2.0.

While the number of unstable multiplications decreased sensibly, cancellations and unstable branchings increased. Using the numerical profiler, we were able to identify that all but one cancellation happened inside Eigen. Eigen [Guennebaud et al., 2010] focuses on numerically stable algorithms (sometimes at the expense of computing speed) and is very aware of those issues [Guennebaud et al., 2018], which is a good reason to believe that, as Shaman tells us, its output is reliable despite the numerous cancellations in the intermediate computations.

This, for us, was the final nail in the coffin for the use of cancellation as a clue to identify the sections of code that produce the numerical error: a large

program is likely to have many cancellations which does not mean that is necessarily numerically unstable. As we could not reduce the final error by improving the numerical stability of the polynomial coefficients computation in Felix 1.0, we believe that the key source of numerical error, which was eliminated in Felix 2.0, was the homemade linear solvers implementations.

8.1.3 Conclusion

We were able to assert that Felix 1.0 had numerical problems that were triggered by two of its test cases, the oscillations in a 1D harmonic potential and the induced fission on a 239 plutonium target. We found that Felix 2.0 is numerically stable on those same test cases. Furthermore, we believe that the improvements in numerical accuracy are due to the introduction of Eigen's linear solvers, replacing the authors own implementation.

Finally, Felix helped us confirm that Shaman works properly on a large code base, at least as well as Verrou, and helped us develop tools to instrument and analyze a code base with limited human interactions. It also led to the development of tagged error as a way to trace the sources of numerical error accurately.

8.2 Validation of a nuclear reaction simulation code

Some programs can have very clear numerical failure points, where one can easily see that the computation returns noise. In this section we study a version of SCAT2000 [Bersillon, 1988] – a code from the world of nuclear simulation which is used to do cross-sections computation using spherical optical model analysis – that has been instrumented with intrusive Chaos polynomials [Sun, 1979], to study the distribution of its outputs as a function of a set of input uncertainties, but showed visible signs of numerical instability for specific input parameters.

Our goal is to assess whether the values obtained with the default inputs are reliable.

8.2.1 Problem statement

SCAT2000 is used to compute cross-sections based on the spherical optical model [Koning and Delaroche, 2002]. It is a well-tested code that has been widely used [Khalaf et al., 2018, Khalaf et al., 2015, Rahman, 2012] and is known to be numerically stable. It was first implemented in Fortran, but we translated it in C++ (using a previous C translation as a starting point and validating our translation with an existing test suite) to be able to instrument it with both intrusive Chaos polynomials and Shaman.

The code takes input parameters describing a system in which a charged particle is projected in an (atomic) nucleus and computes three cross-sections (a cross-section models the probability of a nuclear reaction occurring as a function of the energy of the projected particle): the reaction cross-section, the elastic cross-section and the total cross-section — which is the sum of the previous cross-sections and can be observed experimentally.

The spherical optical model

Schrödinger's equation can be written as follows, where Ψ is the wave function, E is the system energy, $\hbar$ is the reduced Planck constant, m is the particule mass and U is a potential:

$$\nabla^2\Psi(r) + \frac{2m}{\hbar^2}(E - U(r,E))\Psi(r) = 0$$

The spherical optical model is an application of Schrödinger's equation to a potential $U(r,E)$ described as follows:

$$\begin{aligned} U(r,E) = &- [V_v(r,E) + iW_v(r,E)] \\ &- [V_s(r,E) + iW_s(r,E)] \\ &+ [V_{so}(r,E) + iW_{so}(r,E)] \\ &+ V_c(r) \end{aligned}$$

Where the $V_v(r,E)$ is the real volume potential, $W_v(r,E)$ the imaginary volume, $V_s(r,E)$ and $W_s(r,E)$ the real and imaginary surface potentials, $V_{so}(r,E)$, $W_{so}(r,E)$ the real and imaginary spin-orbit potentials, and $V_c(r)$ the Coulomb potential dedicated to the proton interaction description. The first 6 terms are factorized as the product of an energy-dependent well depth and a radial-dependent geometrical factor:

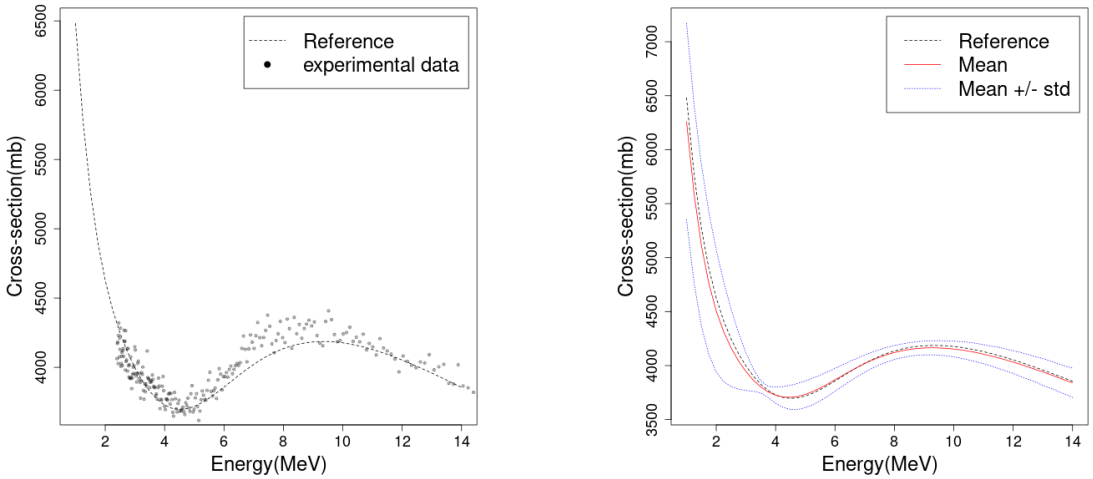
$$\begin{aligned} V_v(r,E) &= V_v(E) f(r, R_{v_v}, a_{v_v}) \\ W_v(r,E) &= W_v(E) f(r, R_{w_v}, a_{w_v}) \\ V_s(r,E) &= V_s(E) g(r, R_{v_s}, a_{v_s}) \\ W_s(r,E) &= W_s(E) g(r, R_{w_s}, a_{w_s}) \\ V_{so}(r,E) &= C_{so} l \cdot V_{so}(E) h(r, R_{v_{so}}, a_{v_{so}}) \\ W_{so}(r,E) &= C_{so} l \cdot W_{so}(E) h(r, R_{w_{so}}, a_{w_{so}}) \end{aligned}$$

The radial form factors $f(r, R_i, a_i) = (1 + e^{-\frac{r-R_i}{a_i}})^{-1}$ $(i = v, s, so)$ are Woods-Saxon types with $R_i = r_i A^{1/3}$, where r_i is the reduced radius (fm). The depth V

(MeV) of each of the six potential wells, the reduced radii r_i and the diffusivities a_i (fm) may have an energy dependence. The parameters of the optical model are r_i, a_i, and V. Consistent sets of parameters are used to calculate physical quantities such as total (σ_T), shape elastic (σ_E), and reaction (σ_R) cross-sections. The cross sections calculated with the spherical optical model can be characterized as a function of the model parameters:

$$\begin{aligned}\sigma(T, E, R) = f(&V_v, V_s, V_{so}, W_v, W_s, W_{so},\\ &a_{V_v}, a_{V_s}, a_{V_{so}}, a_{W_v}, a_{W_s}, a_{W_{so}},\\ &r_{V_v}, r_{V_s}, r_{V_{so}}, r_{W_v}, r_{W_s}, r_{W_{so}})\end{aligned}$$

Figure 8.1a shows the total cross-section (curve) computed for a given input and the corresponding experimental data (dots).



(a) No uncertainty quantification.

(b) Intrusive Chaos polynomials (degree 3).

Figure 8.1: Total cross-section, as exported by SCAT2000, with and without intrusive Chaos polynomials.

Intrusive Chaos polynomials

The spherical optical model relies on a number of physical constants that have been fitted on experimental data. A subset of parameters (r_{V_v}, V_v, a_{V_v}) has been selected using sensibility analysis [Dossantos-Uzarralde and Guittet, 2008] and modeled by a distribution centered on their experimentally fitted values and a standard deviation

representative of the uncertainty on those values. To do so, we instrumented the code with an implementation of Intrusive Chaos polynomials [Sun, 1979] such that, for a given energy, it outputs a stochastic cross-section decomposed on a chosen polynomial basis (Hermite polynomials for a Gaussian distribution and Legendre polynomials for a uniform distribution). The moments (which are visible on Figure 8.1b) can then be recomposed and compared with experimental values.

Intrusive Chaos polynomials get their name from the fact that they rely on replacing every scalar in the computation with a vector representing the decomposition of the value — made stochastic — on an orthogonal polynomial basis. This is done by overloading every operator so that these expansions are properly combined and propagated. Adopting the Le Maître notation [Le Maitre et al., 2004], a parameter θ becomes:

$$\theta(\omega) = \sum_{k=0}^{P} \theta_k \Psi_k(\xi(\omega))$$

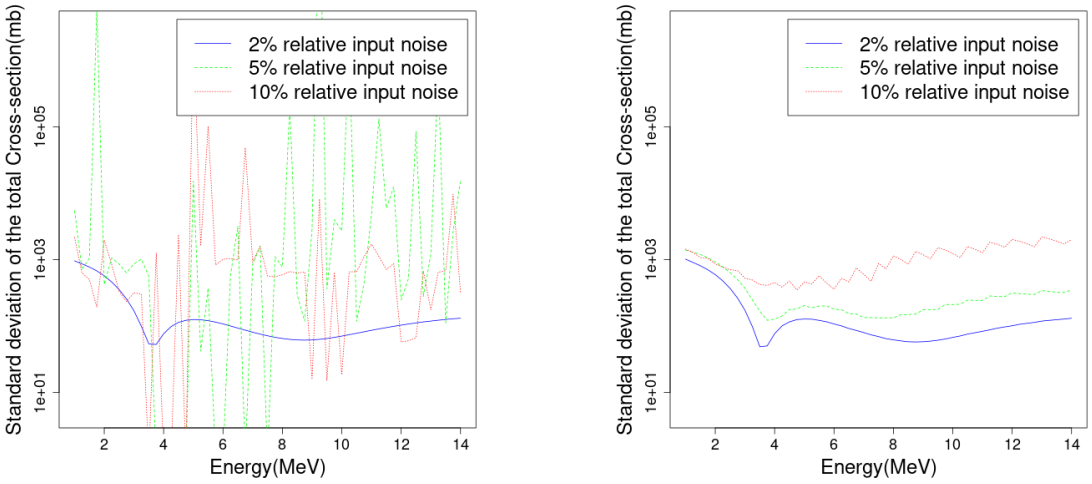
Where $\xi(\omega)$ is a shorthand for a space of random variables $\{\xi_0(\omega), \xi_1(\omega), \ldots\}$ following the same distribution as our stochastic parameter $\theta(\omega)$ and P is the *order of decomposition* of the variable such that $P + 1 = \binom{N+p}{N}$ with p the polynomial truncation and N the number of independent variables made stochastic.

Once instrumented, the code can be thought of as manipulating distributions (encoded as polynomials stored within vectors) instead of scalars, but does so without requiring several runs as a Monte-Carlo approach would. However, intrusive Chaos polynomials require a larger number of arithmetic operations and solving a linear system for each (formerly scalar) division. This casts doubt on the numerical stability of the instrumented program. Using our own, generic, implementation of intrusive Chaos polynomials we were able to investigate this question with Shaman[1].

8.2.2 Evaluation of the numerical error

To make the description of the experiments clearer, we will note *Reference* an experiment done without intrusive Chaos polynomial, and either *Gaussian*(n%) or *Uniform*(n%), an experiment done with intrusive Chaos polynomials modeling a Gaussian or uniform input distribution of n% relative standard deviation.

[1]The code had thus two levels of instrumentation, manipulating vectors of encapsulated errors instead of numbers. It is interesting to note that having that many levels of indirection, plus the template meta-programming introduced by the use of the Eigen library to manipulate the vectors, forced us to use the Clang compiler as the GCC compiler would not terminate when using all levels of instrumentation.



(a) Gaussian distribution.

(b) Uniform distribution.

Figure 8.2: Standard deviation of the total cross-section with inputs of either Gaussian or uniform distribution and an increasing input standard deviation.

The request for a numerical analysis was triggered by the fact that the standard deviation of the output behaved chaotically in the *Gaussian*(5%) and *Gaussian*(10%) cases, as can be seen in Figure 8.2a. Using Shaman we confirmed those plots had no significant digits contrary to the plots made with a uniform distribution (*Uniform*(2%), *Uniform*(5%) and *Uniform*(10%)). The surprising fact about this numerical instability is that using a Gaussian distribution, rather than a uniform distribution, changes the content of a tensor used when one multiplies two numbers (two vectors when using intrusive Chaos polynomials), while increasing the input uncertainty changes one coefficient of the input polynomials. None of these modifications changes the code path and each modification alone (*Gaussian*(2%) or *Uniform*(10%)) results in a numerically stable code.

The final application settled on *Gaussian*(2%): a Gaussian distribution with a 2% standard deviation. A value for which, interestingly, a uniform input distribution and a Gaussian input distribution result in virtually identical output distributions. Shaman was used to show that the numerical error of the output was low enough to make this decision viable.

Table 8.1 shows the mean relative error of each of the three cross-sections in the *Reference* case, which is known to be numerically stable, and *Gaussian*(2%) case. While the error increases by a factor of about 4, which translates in a loss of one significant digit, it stays close to the minimum error one can expect when using double precision arithmetic. This means that, despite going from 2% input standard deviation to 5% input standard deviation is enough to make the

Cross-section	*Reference*	*Gaussian*(2%)
Total cross-section	$2.779260e^{-14}$	$1.023680e^{-13}$
Elastic cross-section	$2.993497e^{-14}$	$9.018840e^{-14}$
Reaction cross-section	$4.847772e^{-14}$	$1.766620e^{-13}$

Table 8.1: Average of the relative numerical error over each cross-section compared between the reference, double precision, program, and the version of the code that was instrumented with Chaos polynomials.

computation numerically unstable and obtain a chaotic output, these values provide a numerically stable result that is viable for further analysis.

8.2.3 Conclusion of the study

The study showed the numerical stability of the program under some hypotheses and was able to confirm that the chaotic behavior of the output was indeed due to numerical errors.

From a more general point of view it is interesting to observe, once more, that a small modification in input parameters can render a code numerically unstable (to the point of behaving chaotically). This sudden drop in precision is non-intuitive, but can be explained by the fact that floating-point numbers and numerical error are encoded exponentially, with powers of two, while we understand numbers on a linear scale. This behavior is slightly concerning since most programs are not validated at a large, production, scale (large scale test cases tend to be slower so one naturally favors smaller, faster, test cases) and one cannot expect all numerically unstable programs to behave chaotically and be easily detectable (such as Felix 1.0, described in section 8.1.2, which had numerical problems despite having been partially validated analytically).

This work was done in collaboration with Pierre Boutry, Pierre Dossantos-Uzarralde, Vincent Nimal, and Thomas Mazurkiewicz who provided a C and Fortran SCAT2000 implementation and the algorithms for the intrusive Chaos polynomial. We did a generic C++ implementation of both SCAT2000 and Intrusive-Chaos-polynomials which, with the result of our study on SCAT2000 and uncertainties, is now pending publication.

8.3 Numerical error and uncertainty quantification

The outputs of a simulation can be impacted by both numerical errors and uncertainties. Some papers will consider numerical error negligible compared to

uncertainty on the inputs (as detailed in section 2.1), but we are not aware of a study actually comparing the two quantities.

In this section, we study the numerical error of a full pipeline, including a physical model (simulating the propagation of sound in an inhomogeneous medium using normal mode decomposition) and various metamodels used to compute its uncertainty. The goal is to check whether the numerical error impacts the model and, going further, whether it impacts the uncertainty quantification.

8.3.1 Problem statement

The program we study uses normal mode decomposition to simulate sound propagation in an inhomogeneous medium [Jensen et al., 2011, Waxler et al., 2017]. A signal is represented as the sum of different wave packets, each one propagating in different waveguides. The waveguides are characterized by the eigenfunctions (ϕ) and eigenvalues (λ) of the operator $L\phi = \frac{d^2\phi}{dz^2} + \frac{\omega^2}{c(z)^2} = \lambda\phi$, where $\omega = 2\pi f$ with f the frequency in Hertz, z the altitude in meters, and $c(z)$ the effective celerity of the wave packet. The operator is discretized as a matrix $\tilde{L} \approx L$, whose spectrum can be computed with classical linear algebra algorithms.

The program was designed to include a further layer which is the study of the behavior of the spectrum when the medium is perturbed. The perturbations are modeled by a random parameter ξ (Gaussian vector of dimension 2) which becomes an input to the function producing the matrix. We study the impact of this perturbation on the eigenvalues of the matrix with a panel of uncertainty quantification methods, going from raw Monte-Carlo to metamodels including non-intrusive Chaos polynomials and Gaussian processess. The computation has three key steps:

- the discretization of the operator L, using a spectral collocation method on a Chebyshev basis, to produce a matrix $\tilde{L}$,
- the computation of the complex eigenvalues of the matrix $\tilde{L}$, using a Schur decomposition, to compute the different contributions to the propagated signal,
- the uncertainty quantification using the algorithms described in section 8.3.3.

We want to compute the numerical error through the full pipeline and assess its impact on the uncertainty quantification methods.

8.3.2 Numerical error and normal mode decomposition

We first measured the numerical error of the physical model, without the uncertainty quantification pipeline.

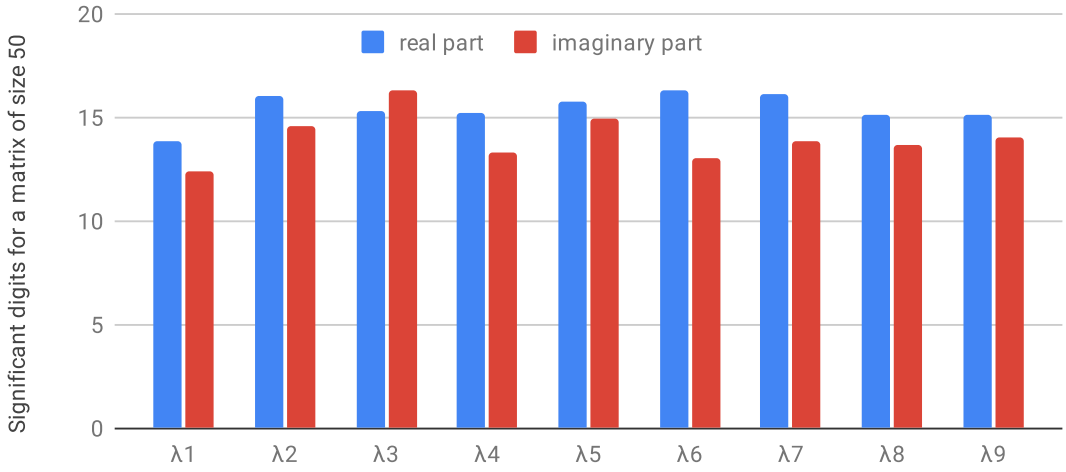


Figure 8.3: Number of significant digits of the eigenvalues of interest for a matrix of size 50.

Figure 8.3 shows the number of significant digits, as measured by Shaman, on the 9 eigenvalues of interest for the model (only a fraction of the eigenvalue produced have a physical meaning) for a coarse discretization (a matrix of size 50).

It can be seen that the number of significant digits is fairly homogeneous from one eigenvalue to the other and that the imaginary parts tend to have less significant digits than real parts. In the following we will average the number of significant digits over all eigenvalues of interest in order to simplify plots (this is meaningful as they have a similar precision).

Figure 8.4 illustrates the evolution of the average number of significant digits with matrix size (larger matrices are associated with a finer discretization and, theoretically, a smaller discretization error). One can still see that the imaginary part has less significant digits but, more important, the number of significant digits decreases with the matrix size and starts significantly dropping irregularly when the matrix gets roughly larger than 250 rows.

It is important to note that this drop in significant digits is a particularity of the operator L and happens during its discretization (as can be seen in Table 8.2). It is not a property of eigenvalue decomposition in general and we observed much more robust numerical behavior with other operators.

The standard deviation between realizations of an eigenvalue (eigenvalues sampled from different perturbations) is an important metric for uncertainty quantification. To study the evolution of the precision of the standard deviation as a function of the size of the matrices, we drew one hundred matrices and computed the standard deviation of each eigenvalue of interest across the samples. Figure 8.5

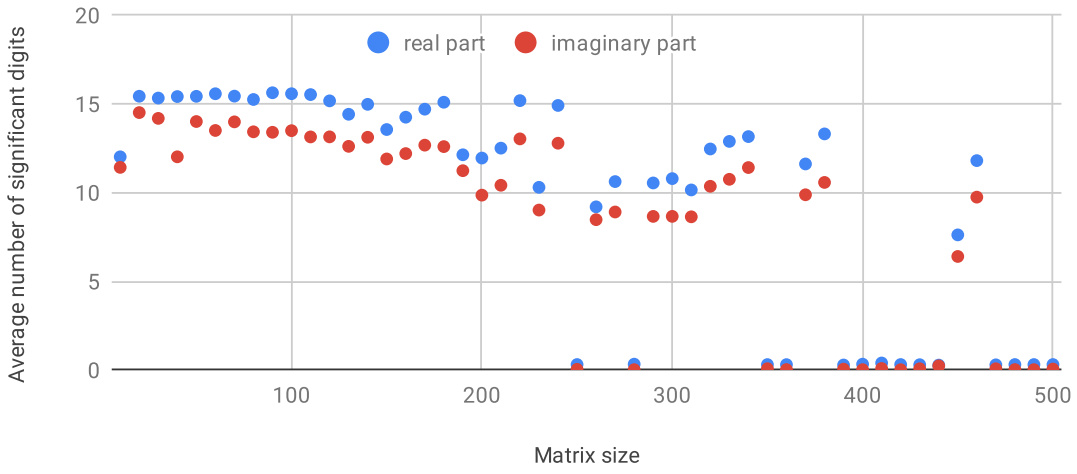


Figure 8.4: Average number of significant digits of the eigenvalues of interest as a function of the matrix size.

gives the numbers of significant digits of those standard deviations averaged over the eigenvalues of interest. One can see that the standard deviation suffers much sooner from the drop in significant digits (around matrices of size 160), probably because it can get tainted and deeply impacted by even a single eigenvalue with zero significant digits.

This behavior is concerning as it highlights the fact that the uncertainty quantification could be much more fragile than the original computation as it combines outputs. The next section checks whether this truly happens with a panel of uncertainty quantification methods applied to the code.

8.3.3 Uncertainty quantification

Methods

Uncertainty quantification methods are used to study the impact of the uncertainty of an input on the output of a program. This is done by modeling the uncertainty on the input with a stochastic distribution and studying the distribution of the output (usually its standard deviation as a way to quantify its spread).

Monte-Carlo The Monte-Carlo method [Bipm et al., 2008] consists of doing the computation a large number of times with random input noise drawn from a distribution that represents our uncertainty on the inputs. Given enough samples, the output standard deviation will reflect the impact of the input noise (and thus

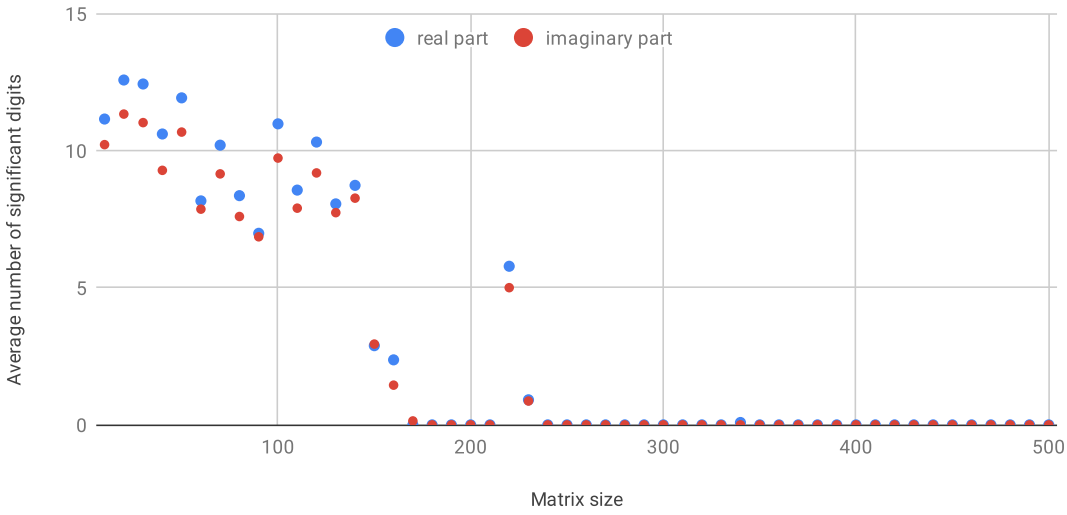


Figure 8.5: Average number of significant digits of the standard deviation of the eigenvalues of interest as a function of the matrix size.

parameter uncertainty) on the output.

The strengths of this method are that it is fairly straightforward to implement, impervious to the dimension of the problem (the Monte-Carlo method was first designed to compute high-dimensional integrals), and trivially parallelizable. However, it can require a very large number of samples to converge (convergence in $O(\frac{1}{\sqrt{n}})$, where n is the number of points), which is a problem when running the program once already takes a non-trivial amount of time.

Gaussian processes Gaussian processes [Williams and Rasmussen, 2006] take a number of (input,output) pairs in order to model a system. The following formula gives us $P(y^*|x^*, f(x), x)$, the probability of observing a given output y^* given training pairs $(x, f(x))$ and some new inputs x^*, which is assumed to be a Gaussian G:

$$\begin{aligned} P(y^*|x^*, f(x), x) &= G(y^*|\mu, \sigma^2) \\ \mu &= K(\omega, x^*, x)K(\omega, x, x)^{-1} f(x) \\ \sigma^2 &= K(\omega, x^*, x^*) - K(\omega, x^*, x)K(\omega, x, x)^{-1}K(\omega, x^*, x)^T \end{aligned}$$

This works under the hypothesis that the function modeled, f, follows a zero-mean Gaussian distribution, in a space with one dimension per input point, with a given covariance K (which has some hyperparameter ω). Using this model requires

selecting an appropriate covariance function (also called a kernel) and fitting the hyperparameters (usually by maximization of the log-marginal likelihood).

Here they are used as a metamodel for our physical simulator: we train the Gaussian process on a handful of samples to emulate the normal mode decomposition. We can then do a Monte-Carlo analysis on the metamodel which can be evaluated much faster than the physical simulation.

One strong point of Gaussian processes is that they provide an uncertainty on their output which can be propagated in the uncertainty quantification to take the (meta-)modeling error into account. It can also be used to select samples cleverly (no need to compute samples for an area of the input space that is already properly modeled).

Their main weak point is that one needs to select a kernel, a measure of proximity between inputs, appropriate for the domain under study and to fit some hyperparameters making this method non-trivial to apply properly.

Non-intrusive Chaos polynomials Non-intrusive Chaos polynomials [Wiener, 1938] are another kind of metamodel, this one based on polynomial regression on a basis of orthogonal polynomials chosen to match the distribution of the input noise. Contrary to intrusive Chaos polynomials (as seen in chapter 8.2), they do not require instrumenting the program. Instead they model the output y as a polynomial of an input ω and a perturbation ξ:

$$y(\omega, \xi) = \sum_{k=0}^{P} \beta_k(\omega)\Psi_k(\xi)$$

The polynomial, of degree P, is expressed as a sum of orthogonal polynomials Ψ_k weighted by coefficients β_k that can be fitted either with arbitrary points by generalized regression or using carefully selected quadrature points associated with weights. In the following we will cover both methods, using Gaussian quadrature points.

Non-Intrusive Chaos polynomial have the advantage of being fairly straightforward to use and not requiring a Monte-Carlo analysis in order to get the final standard deviation as it can be deduced from the coefficients of the polynomials.

Their main weakness is that the polynomial basis grows quickly with the number of inputs making them unwieldy for problem with many inputs, and that they are restricted to smooth function (due to their polynomial nature), making them unsuitable for some tasks.

Impact of the numerical error on the uncertainty computation

In order to quantify the resilience of the uncertainty quantification methods to the numerical error, we measured the average number of significant digits of the expected value of the eigenvalues of interest according to the various models, as can be seen in Figure 8.6.

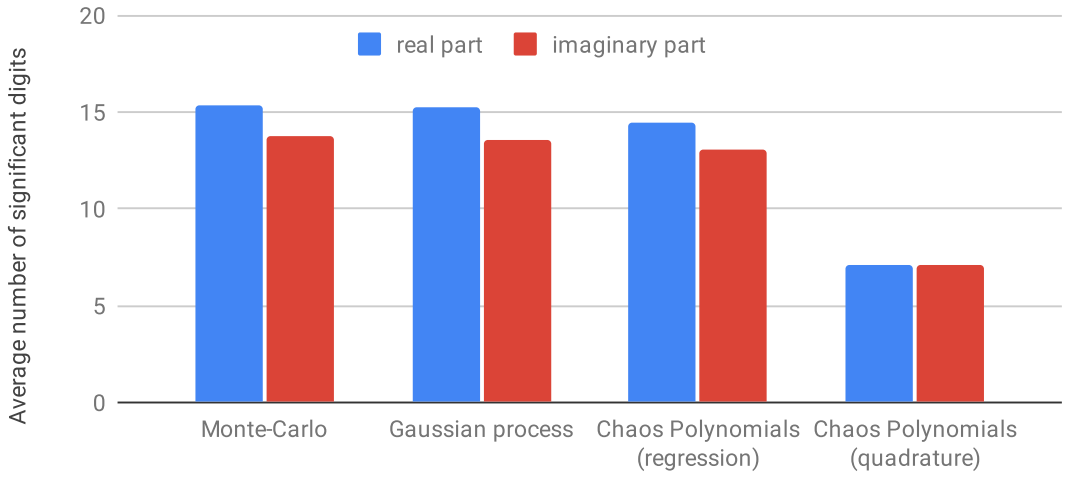


Figure 8.6: Average number of significant digits as a function of the uncertainty quantification method for a matrix of size 50.

It is interesting to note that, for a matrix of size 50 (small enough to insure that the computation of the eigenvalues is numerically stable), the metamodels are as precise as the Monte-Carlo method except for the Non-Intrusive Chaos Polynomials fitted on quadrature points which are noticeably less precise. This can be explained (and was indeed confirmed using tagged error) by cancellations in the integration used to compute the polynomial coefficients, a phenomenon typical to quadrature methods that can be alleviated with carefully designed code combined, often, with higher precision.

When we get to matrices of size 200 (at which point we know that the numerical error would be significant in the standard deviation, but not the eigenvalues), as can be seen in Figure 8.7, the precision of all metamodels degrades significantly and the quadrature method stops being an outlier. This can be explained by the fact that metamodels will try to maximize the information extracted from a small number of points. This means that they can, as feared, be tainted and invalidated by some points with low precision. This hypothesis was confirmed with tagged error which showed that the numerical error was transmitted from the operator discretization and not generated by the metamodels themselves as can be seen in

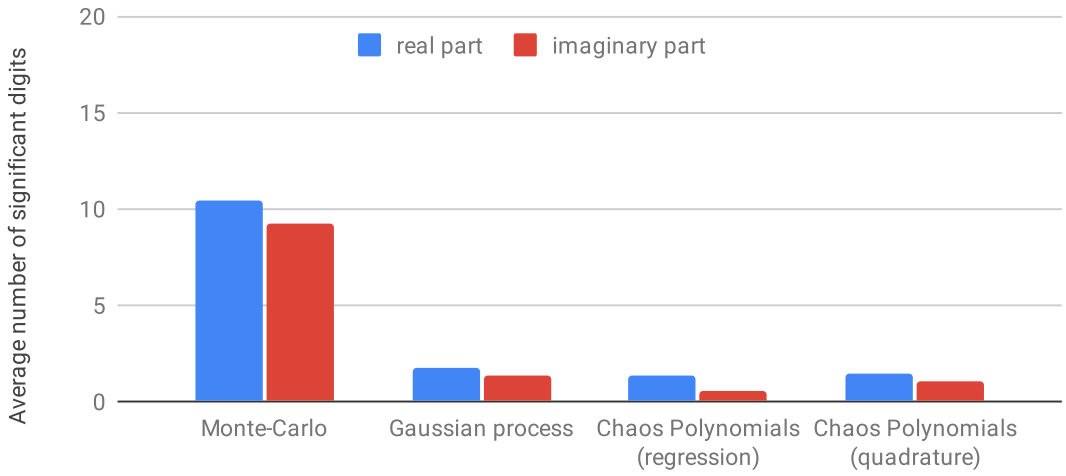


Figure 8.7: Average number of significant digits as a function of the uncertainty quantification method for a matrix of size 200.

Table 8.2. Meanwhile, the Monte-Carlo method remains accurate. This is probably because it relies on a simple average and is thus not likely to be impacted by non correlated numerical errors.

Matrix size	Operator discretization	Metamodel fit
50	$\leq 1\%$	99%
200	99%	$\leq 1\%$

Table 8.2: Main sources of numerical error, in percentages, when computing eigenvalues with Chaos polynomials fitted by quadrature. The percentages are averaged over fifty samples and over all eigenvalues.

8.3.4 Conclusion of the study

Regarding this program, we learned that the numerical error is fairly homogeneous between eigenvalues and that it grows abruptly with the matrix size. In particular, eigenvalues stop being reliable for matrices of size larger than 250 and uncertainty quantification is impacted past 160 rows.

A more general conclusion is that, by their very nature, metamodels are particularly fragile when they receive bad inputs. The Monte-Carlo methods, despite being slower, has the advantage of being very resilient to such problems.

This work was done in collaboration with Alexandre Goupy who developed the core application and did the computations related to the uncertainty quantification. The study also required the ability to map two sets of eigenvalues generated with different input noise (a naive nearest neighbor mapping gives misleading results for this use case). We designed an algorithm to do this mapping which is not given here. We are now writing the full, detailed, analysis of the code for publication.

Part IV

Predicting the convergence profile of a linear solver

Table of Contents

Predicting the convergence profile of a linear solver

9.1 Introduction

This part of the dissertation is set aside as it does not directly concern numerical error. Rather, it focuses on the compromise between precision and performance.

Linear systems appear in all forms of simulation, but as there is a large variety of solvers and preconditioners tuned for specific needs, selecting an appropriate solver/preconditioner pair for a new, large, linear problem is still more of an art than a craft. This leads to conservative decisions and preventable slow convergence.

We propose an application of machine learning to tackle this problem. In high-performance computing, machine learning is often suggested to replace computationally expensive simulations to make them run faster at the risk of getting absurd (or worse, good-looking, but misleading) results. In the following sections, we suggest a different use of machine learning, not to replace code, but for decision support. Making the computation faster, but also more accurate, while still using a proven algorithm[1].

The first section introduces the linear solvers and preconditioners covered by this study. The second section does a short review of the types of method that are used to select an appropriate linear solver. The third section describes our solution and the associated algorithm in detail. Finally, we review the quality of the results on some randomly drawn examples.

9.2 Linear solvers and preconditioners

In this section we introduce the panel of linear solvers and preconditioners that will be covered by our work.

[1] A video, introducing this work at the *2020 Digital French-German Summer School with Industry*, is available online [Demeure, 2020].

Linear system A linear system is an equation of the form $Mx = b$ where M is a known matrix, b a know vector, and x an unknown vector which we want to determine. The system can also be rewritten $MP^{-1}Px = b$ or $P^{-1}(Mx - b) = 0$ if one introduces a right or left preconditioner, P, to make it easier to solve. Once solved, the quality of a given solution, x, is usually measured with its residual $||Mx - b||_2$ or the relative residual $\frac{||Mx-b||_2}{||b||_2}$, which has the advantage of being normalized and comparable between different linear systems.

Linear solvers An iterative solver takes an initial solution, x_0, to a linear problem and improves it iteratively until it reaches a convergence criteria, typically a condition on the residual of the problem. They are of particular importance to solve large sparse numerical problems that cannot be resolved in a reasonable time with direct solvers (such as the LU factorization).

In our work we experiment with the following eleven Krylov-based solvers from the Belos library [Bavier et al., 2012]: *Block CG, Pseudo-Block CG, Seed CG, Recycling CG, BICGSTAB, Block GMRES, Pseudo-Block GMRES, Recycling GMRES, MINRES, Pseudo-Block Transpose-Free QMR, Transpose-Free QMR.* As some of those solvers have prerequisite (i.e., CG, the conjugate gradient algorithm requires a symmetric positive-definite matrix), whenever we test solvers on a problem, we check whether their working hypothesis are satisfied.

Preconditioners One can also, optionally, use a right or left preconditioner, adding some computations to try and reduce the condition number of a problem which might speed up the convergence of the solver. We will cover the following eight preconditioners from the IfPack2 library [Prokopenko et al., 2016]: *ILUT, RILUK, DIAGONAL, CHEBYSHEV, Richardson, Jacobi, Gauss-Seidel, Symmetric Gauss-Seidel.*

Trilinos Belos and Ifpack2 are both part of the Trilinos project [Heroux et al., 2005], a collection of state of the art linear algebra libraries that are interoperable and compatible with CPU parallelism, MPI, and CUDA. Our work could be done with any linear algebra implementation, but we decided to focus on Trilinos as it is a production implementation commonly used to deploy large scale simulations on supercomputers, and not a toy implementation. Thus, showing that our work is applicable to non-trivial implementations and could be integrated into a production system.

9.3 Selecting a linear solver

Selecting the proper solver and preconditioner pair for a linear system is a combinatorial problem. Using only the linear solvers covered in this study and, optionally, a right or left preconditioner, there are already $11 * (1 + 2 * 8) = 187$ possible combinations. However, selecting a good linear solver can produce more accurate results faster, hence there is a need for a method able to help with this selection.

This section does a quick review of the approaches currently used to select a linear solver for a given linear system and explains their shortcomings.

9.3.1 State of the art

Heuristics The classical approach, and one that is encoded in some linear algebra systems such as Matlab, is to make a decision based on the properties of the linear system.

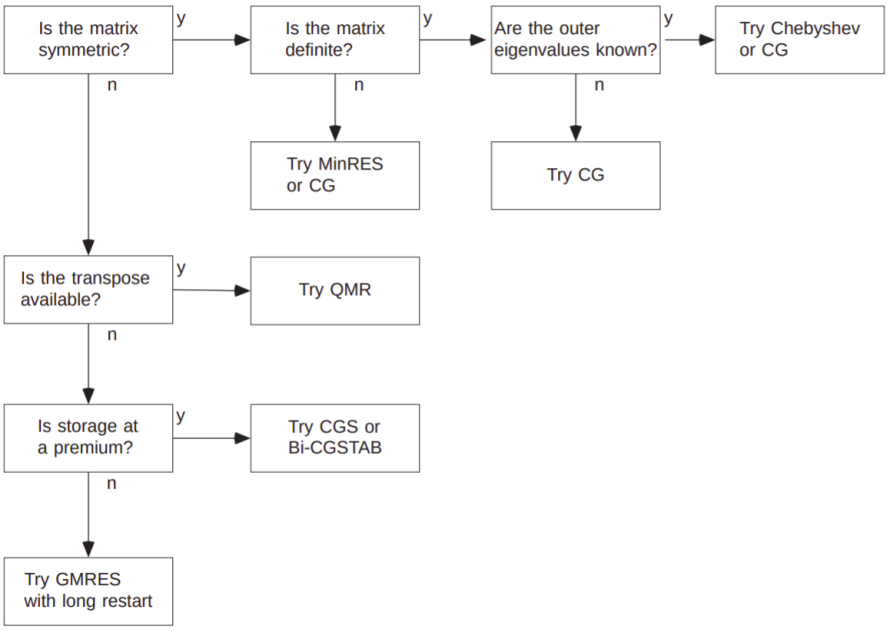


Figure 9.1: Flowchart for the selection of an iterative method. Picture extracted from [Barrett et al., 1996].

Figure 9.1, taken from [Barrett et al., 1996], illustrates such an heuristic. The main limit of this approach is that it is very coarse grained, you will note that it covers only the solvers (and not the preconditioners), a fraction of our 187 possibilities and that the diagram indicates that the user might want to *try* one

or more possibilities. This stems from the fact that very little problem-specific information is used to select a solver.

Machine learning As is often the case when a problem is described as a matter of human experience and intuition, researchers suggested the use of machine learning to select a linear solver and preconditioner.

In particular, J. Yeomn, J. J. Thiagarajan, A. Bhatele, G. Bronevetsky and T. Kolev in their paper [Yeomn et al., 2016] whose research serves as a foundation for our work. We will reuse their dataset and build on the features they used to describe linear systems.

However, their model predicts which linear solver would converge first for a given linear system and *not* the final residual. This poses a fundamental problem as a linear solver might converge very quickly to a solution that is not accurate enough for a given application. Furthermore, one might want to wait a little longer if it means that the result would be noticeably improved. This led to the development of our method.

9.3.2 A multi-objective problem

Selecting a linear solver and preconditioner for a linear system is a multi-objective problem. A typical user will have both a time budget and a minimum viable accuracy. As both computing time and final accuracy matter, one can plot the trade-off as a convergence profile: a plot with the computing time on the x-axis and the residual on the y-axis.

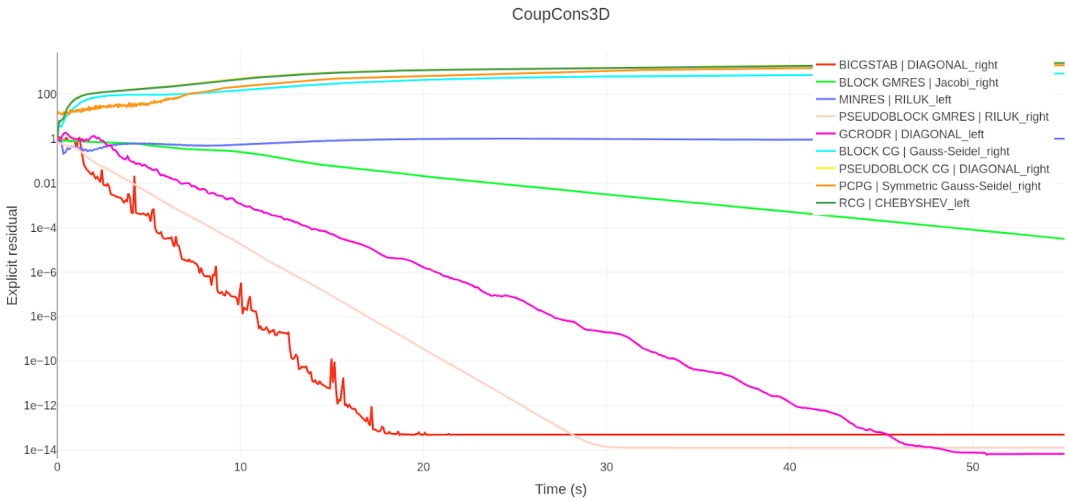


Figure 9.2: Convergence profile of a random selection of linear solvers and preconditioners for the CoupCons3D problem. The relative residual is displayed on a log scale.

Figure 9.2 shows one such plot. BICGSTAB converges first, but ten second later, PSEUDOBLOCK GMRES converges to a solution with a ten times smaller residual. Note that the y-axis features a relative residual (normalized between linear problems) and that we chose to stop linear solvers at 2000 iterations (enough for 99% of solvers to consider that they had converged on our dataset) and not a given time which explains why not all plots have the same length.

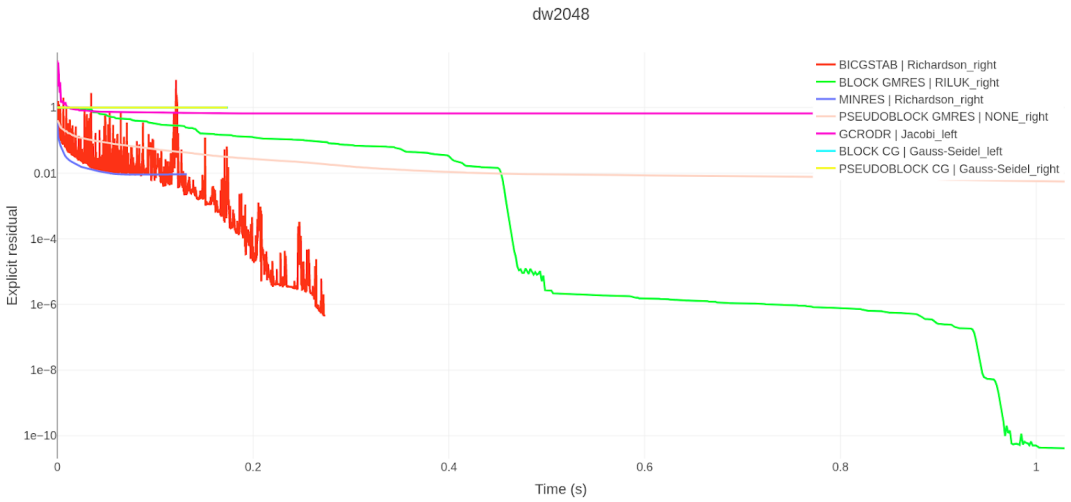


Figure 9.3: Convergence profile of a random selection of linear solvers and preconditioners for the dw2048 problem. The relative residual is displayed on a log scale.

Figure 9.3 shows a more extreme example where computing time and final precision vary widely between linear solvers. Given such a plot, one could handpick the linear solver that is the best fit for one's application. This raises a natural question: can we predict those convergence profiles with enough accuracy to help a user take an informed decision?

9.4 Our model

In the following, we propose a model using cheaply computed features to predict the convergence profile of a linear solver on new problems. This let us recommend a solver and a preconditioner as a function of the time and accuracy budget of a user. Our solution is based on artificial neural networks [McCulloch and Pitts, 1943]. A common machine learning algorithm that is well suited to the production of a plot, instead of a single scalar.

9.4.1 Structure

General structure We want to plot the residual for the first 2000 iterations of each solver/preconditioner pair as a function of the computing time which means that our plot will have a variable length.

To solve this problem, one could use methods from the field of time series regression, but we opted for a simpler solution: building two neural networks and

combining their outputs.

Both models take features describing the linear system / solver / preconditioner triplet as input. The first model has 2000 outputs and predicts the relative residual for the first 2000 iterations. The second model outputs the logarithm of the mean computing time per iteration which we consider constant during the computing time (as a first approximation).

Using the output of the second model, one can plot the output of the first model as a function of the computing time and plot the convergence profile.

Neural network While the prediction of the mean computing time could probably be done by a simple model (it is mostly a polynomial function of the size of the matrix with solver and preconditioner specific coefficients), the residual curves is a much more complex matter. We did some preliminary tests with both Gaussian processes [Williams and Rasmussen, 2006] (which would have given us uncertainty estimates) and neural networks, but settled on the latter as they gave a mean absolute error two to ten times smaller for this problem.

As we were using neural networks, we decided to standardize our architecture and use the same network structure for both models in order to reduce the code complexity, namely a fully connected deep neural networks with four inner layers of size 2000, 1000, 500, and 500 and an output layer of size either 1 or 2000, depending on the model. What we call a layer is the composition of a linear layer, batch normalization [Ioffe and Szegedy, 2015] and a ReLU activation function [Glorot et al., 2011].

The number and size of the inner layers have been selected manually using commonly used defaults as a starting point. It seems important to note that, in our experiments, whenever we added hundreds of matrices (which translate into tens of thousands of samples), we were able to increase the size of the network and improve on our previous results.

Some inputs, such as the linear solver and preconditioner, are categorical (and thus non-numeric). We used embeddings [Bengio et al., 2003] to deal with them as detailed in Section 9.4.3.

It is important to note that this is a classical, well tested, non-surprising network structure. We did not try to introduce a new domain-specific highly-tuned architecture, but rather to build on a structure that is known to be solid.

9.4.2 Dataset

Inspired by [Yeomn et al., 2016], we took our linear problems from the SuiteSparse Matrix collection [Davis and Hu, 2011]. We used 1500 real square matrices, setting 20% aside to build a validation set. Each matrix produces one sample for each

solver/preconditioner pair, but we set matrices and not just samples aside to avoid data leakage between solvers that behave similarly for the same matrix.

The matrix features, described in Section 9.4.3 were computed with a Julia script. As the features are fast to compute, it takes less than one hour on a desktop to gather the features for all matrices from the dataset (less than one second per matrix).

The value of the initial solution of the systems was set to zero and their second member generated randomly by sampling uniformly between 0 and the infinite norm of the matrix.

We then solved each linear system for all combination of solver, preconditioner and preconditioner side twice. Once to collect the runtime and a second time to collect the residual plot as we use the explicit residual (and not the, very approximate, implicit residual), which is not always available in linear solvers and thus require additional computation which impacts the measure of the computing time. Collecting those data is a highly parallel task which took weeks of computation on a supercomputer.

9.4.3 Features

Our features are inspired by the features described under the heading *basic features* in [Yeomn et al., 2016] to which we added approximation of features linked to the condition number computed with the diagonal elements of the matrix instead of its spectrum.

We applied feature-specific transformation (that will be detailed in the following subsections) and, when appropriate, normalized the features by subtracting their mean and dividing by their standard deviation.

Finally, we selected the best performing subset of features using Bayesian optimization in order to prune features that led to models prone to overfitting (memorizing the data without generalizing to examples out of the training dataset).

Categorical features Categorical features are described using embeddings [Bengio et al., 2003], a technique from the world of natural language processing that consists in learning a vector representation for each category in the process of training the network.

The solver and preconditioner are fully described by three categorical features: the *name of the solver* (embedding of size 6), the *name of the preconditioner* (embedding of size 8), and the *preconditioner side* (none / right / left) (embedding of size 3).

We use only one categorical feature in the description of the linear problem: whether the system is positive definite or not (embedding of size 2).

Normalized by matrix size Some features were normalized by taking the size of the matrix into account. The *number of diagonal dominant rows*, the *maximum number of non-zero per row*, the *lower bandwidth*, and *upper bandwidth* are proportional to the number of rows of the matrix, and thus, normalized by dividing them by the number of rows. The *number of non-zero in the lower triangular matrix* is proportional to the number of elements in the matrix, and therefore, divided by the number of rows squared.

Log-scale We take the logarithm of the absolute value (when needed) of some features to put them in log-scale. Those features are the *number of rows*, the *norm 1* of the matrix, the *frobenius norm* of the matrix, the *approximate maximum eigenvalue*, the *approximate second maximum eigenvalue*, and the *approximate minimum eigenvalue*.

Note that we do *not* compute the eigenvalues, they are approximated using the diagonal of the matrix. While it is a rough approximation, those features were reliably preserved by Bayesian optimization when selecting features, meaning that they still contains rich information on the linear system.

During our tests, it was interesting to observe that whenever we use the raw number of row instead of a log scaled version, the model would instantly use it to overfit.

Already normalized Finally, some features are normalized by construction and already within [0,1]. Those features were left untouched: the *upper spread*[2], the *lower spread*, the *symmetric score*, the *skew symmetric score*, the *pattern symmetric score*, the *approximate s90*, the *approximate s10*, and the *approximate smid*.

s90, s10 and smid are the fraction of singular values in an interval between the lowest singular value and, respectively, 90%, 10%, 50% of the value of the largest singular value. As with the eigenvalue, we do *not* compute the singular value and compute those metrics on the square root of the absolute value of the diagonal elements instead.

9.4.4 Training

The models were trained for 20 epochs using batches of size 512, the AdamW optimizer [Loshchilov and Hutter, 2017], a 10^{-3} weight decay, a momentum of 0.95 and 0.85 respectively, and a cyclical learning rate [Smith, 2017] going from 3×10^{-3} to 0 with a peak at 10^{-2}. Those values have been obtained by a mix of manual experimentation and Bayesian optimization.

[2]The spread is the normalized sum of distance from diagonal to non-zero elements.

We used the *root mean square error* (RMSE) as the metric optimized when fitting on the computing time and the *mean absolute error* (MAE) when fitting the residual plots. This latter choice is important, not only does the MAE make intuitive sense for this use case (as it can be interpreted as the difference of area between the target curve and the curve produced by the model), but using the RMSE (the classical error metric for most problems) resulted in a model that had difficulties converging. We hypothesize that those difficulties stem from the RMSE focusing on the worst predictions (which might not be improvable due to the random nature of the target objective), due to its quadratic nature to the detriment of the overall performances.

9.5 Results

Once trained, the model reaches a root mean square error of **0.4** for the prediction of the logarithm of the mean runtime and a mean absolute error of **4.5** for the prediction of the residual curves.

One problem with those numbers is that they are raw numbers, hard to interpret without points of reference such as prior art that would try to solve the same problem with the same metrics. Thus, the following paragraphs contains predicted convergence profiles and compares them with the experimental data gathered on the same linear problems / solvers / preconditioners combination to check whether the algorithm provided enough information to take an informed decision.

The plots have been drawn at random amongst linear problems where there exists at least one solver/preconditioner combination that converges. For each linear system we applied all possible linear solvers and, for each, one random solver and preconditioner. This was done to mitigate the fact that displaying all preconditioner and solver combinations makes the plots unreadable (see Figure 9.7 for a figure displaying all possible preconditioner for a single linear solver). In the plots, the dashed line represents the prediction and the continuous line the actual plot as recorded when solving the problem.

Solver selection Figure 9.4 shows a typical output. It is *not* a perfect fit as can be seen on the GCRODR solver in particular (plotted in magenta), but we *still* recover the information needed to make a decision. Namely, the fact that there are three clusters of solver: the solvers that do not converge (with a relative residual of one or more), a solver that converged slowly and to an inaccurate value (the GCRODR solver which, in fact, diverges after reaching the predicted minimum residual), and solvers that converge quickly and to a very small residual. While the plots are not perfectly matched, the prediction exhibits the three clusters,

associating them with the correct solvers, and gives an accurate magnitude for the final residual of the solvers.

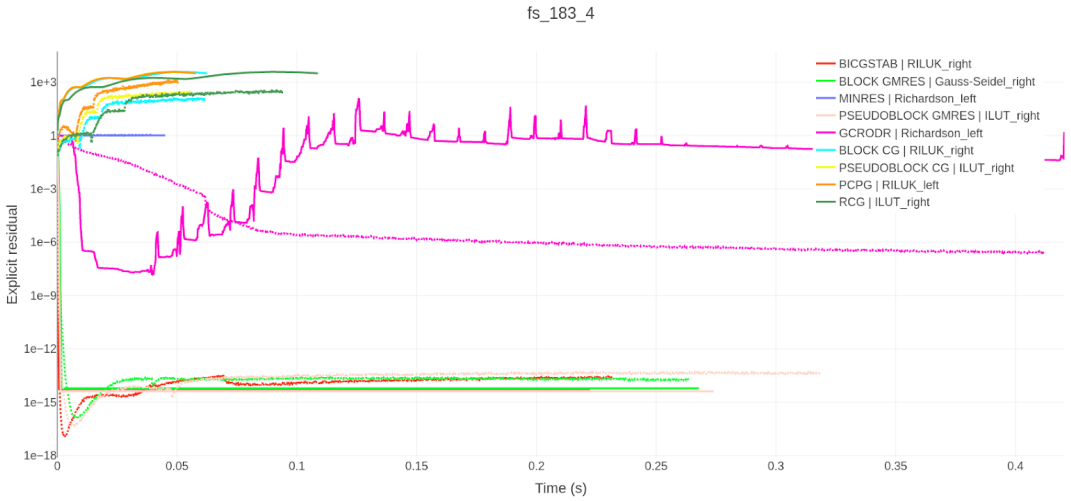


Figure 9.4: Convergence profile of a random selection of linear solvers and preconditioners for the fs_183_4 problem.

Figure 9.5 shows a problem where the various solvers have very different computing time. Once more, we observe that the final residual is accurately predicted along with the speed at which it will be reached. The fast solvers are also properly characterized.

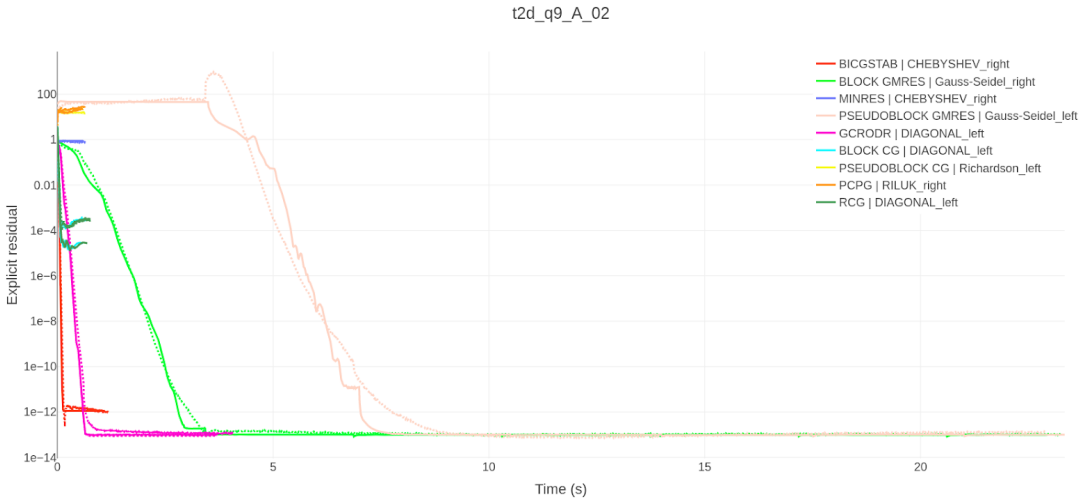


Figure 9.5: Convergence profile of a random selection of linear solvers and preconditioners for the t2d_q9_A_02 problem.

Figure 9.6 shows an extreme failure case. The algorithm predicts that no solver will converge here (amongst the selection that we displayed), while there is a solver that did converge. This failure case is mitigated by the fact that we display only a subset of solver/preconditioner combinations, the probability of such a failure is much smaller when considering all 187 combinations. Overall, most of the failure case we observed led to the selection of a converging but non-optimal solver.

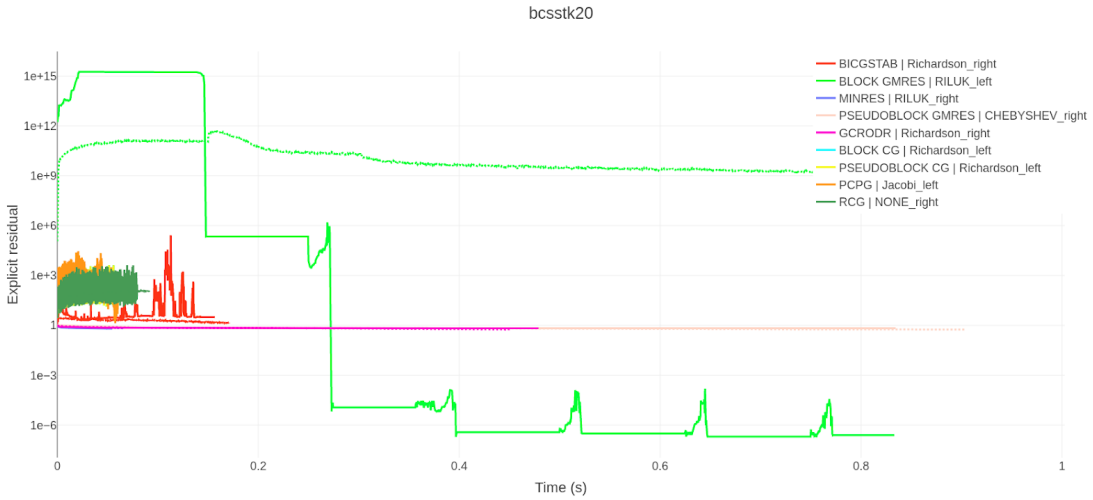


Figure 9.6: Convergence profile of a random selection of linear solvers and preconditioners for the bcsstk20 problem.

Preconditioner selection In this section, we fixed both the linear system and the linear solver in order to focus on the selection of the preconditioner and preconditioner side.

Figure 9.7 shows all possible preconditioners and preconditioner sides for a given solver. The prediction exhibits three clusters that converge to roughly the same final accuracy, but at varying speeds. When compared with the experimental results, we see that the prediction of the final accuracy and relative speed of the preconditioners are accurate. The only noticeable mistake is that some very fast solvers end up in the central, moderately fast, cluster.

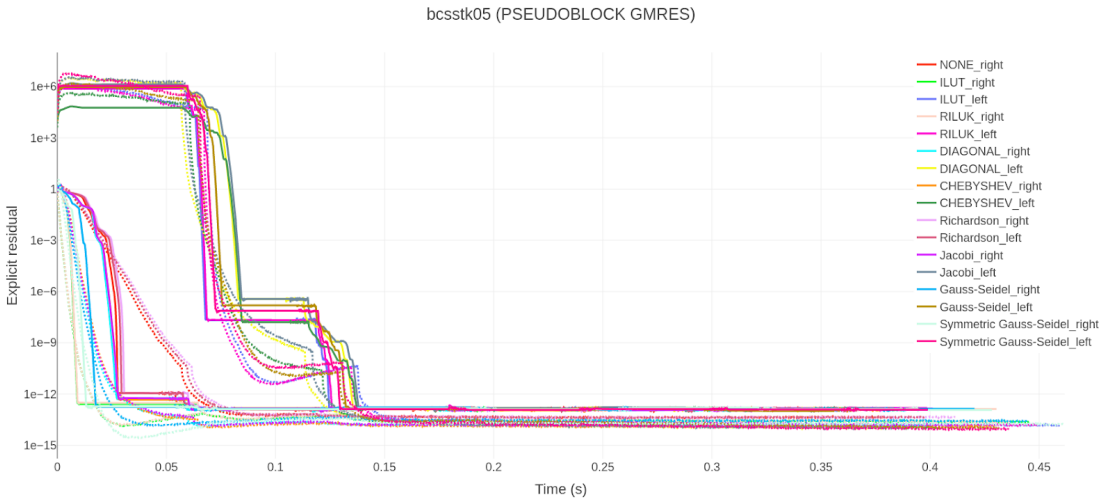


Figure 9.7: Convergence profile of all preconditioners with Pseudoblock GMRES on the bcsstk05 problem.

Figure 9.8 shares some similarity with Figure 9.6 in that there is only one preconditioner that is able to get the given solver to converge for the problem (which highlights the importance of selecting an appropriate preconditioner). Here, however, the prediction is accurate and leads the user to select the appropriate preconditioner.

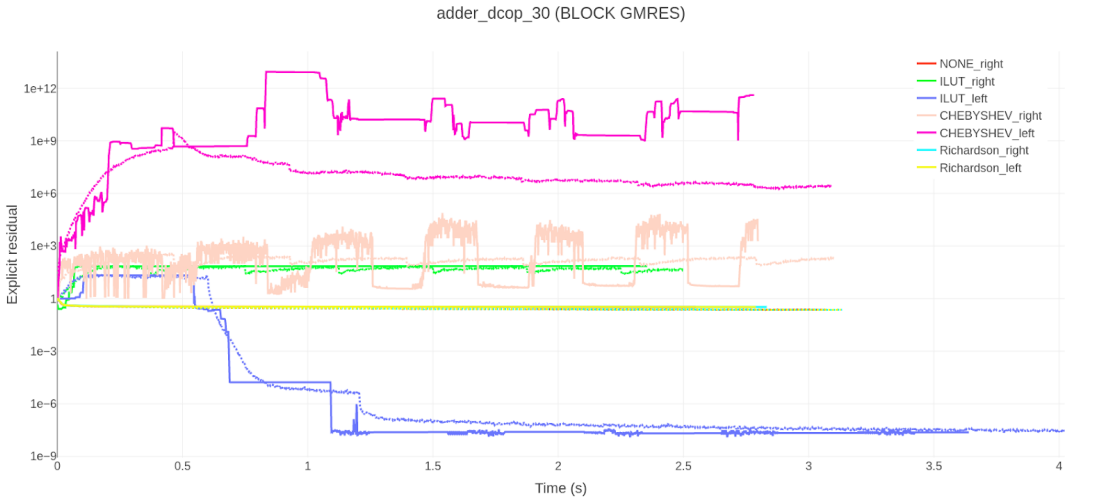


Figure 9.8: Convergence profile of all preconditioners with Block GMRES on the adder_doop_30 problem.

9.6 Overview and perspectives

9.6.1 Overview

First and foremost, this work is a proof of concept and not a finished tool. It is an exploration of the possibility of extracting rich information on the behavior of a linear solver for a given, not observed before, linear system from simple features.

From this point of view, this experience was a success. Instead of a single metric, such as a runtime, we manage to predict the full convergence profile. Information rich enough to help a user trade-off accuracy and precision explicitly in their choice of linear solver and preconditioner. Doing so will improve both the computation speed and accuracy of the result.

Our main takeaway would be that data matters more than the architecture of the model. Our first prototypes built on a fraction of the final training set returned barely usable results, but we observed significant improvements every time we increased the size of the dataset. Those trumped all improvement incurred by the introduction of more complex / clever architectures which led us to keep a fairly simple and straightforward architecture as this was not the bottleneck for the accuracy of our outputs.

Finally, it seems important to insist that we are trying to augment existing algorithm. We are *not* trying to replace existing linear algebra solvers with machine learning, but rather to select appropriate solvers for given problems. This means that, while having benefits in computing speed and accuracy, in the end the system is solved by a genuine linear solver that has been proven to work. This concept of augmenting rather than replacing seems, for us, to be a viable path to integrate machine learning with computations, such as physics simulation, where one cannot afford the fragility and black-box aspect of machine learning.

9.6.2 Perspectives

Of course, one could introduce more specialized architectures, such as models from the world of time series. However, as said previously, this is currently not the bottleneck for the accuracy of our predictions and we expect that adding data would both be an easier and a more efficient way to improve the results.

A very simple way to improve the accuracy of the model would be to train it on matrices from a more specialized domain. The SuiteSparse Matrix Collection covers a very wide panel of use cases and matrix profiles, but reducing the variances in our input by focusing on a domain of interest, would make the problem much easier to learn.

Another, more complex, way to improve the usefulness of the results would be to add uncertainty estimation. Using Bayesian methods, one could associate

uncertainty bars with our prediction which would be helpful to properly take risks into account when deciding on which solver to use. This comes back to our focus on decision support: helping a user to make a decision with rich information, rather than maximizing a metric.

Finally, one of our goal when we started this project (which was set aside due to time constraints) was to first predict the residual, but then gather data to predict the numerical error in the output of the linear solver as a function of the number of iterations. This quantity is orders of magnitude slower to collect, but as it should be correlated with the residual, one could collect only a small dataset and use transfer learning to retrain the network that was used to predict the residual plot for this new task.

Conclusion

Conclusion

We started this work wanting to develop methods to help developers of numerical computations make informed decisions about the trade-offs between precision and performance. The possibility to tweak precision in order to improve performance is getting increasing attention now that Moore's law is reaching its limits and one cannot expect to get significant performance improvements from new hardware. This creates a need for methods to study precision and performance in high-performance computations. Most of this dissertation focused on a way to measure the numerical accuracy of a program and, if an instability is detected, find the sources of numerical error within that program. This is the work that I will review here. I set aside our work on predicting the convergence profile of a linear solver as it is of a different nature. However, as is said in section 9.6, now that we have shown that one can take very simple features and produce predictions accurate enough to improve both the run time and precision of a result by selecting an appropriate linear solver and preconditioner, I do believe that this area of research deserves further study.

Developers who want to trade precision for performance have a vast array of tools to measure performance but, there is a lack of tools to measure precision in a high performance environment. As we described in Chapter 3, high precision arithmetic, interval arithmetic, stochastic arithmetic, and local analysis have all been used to measure the numerical accuracy of a program. However, none of those methods combine an overhead low enough to be applied to long running computations, accuracy over a large number of operations, compatibility with parallelism, and results interpretable by a user who is a domain expert (such as a physicist) rather than a floating-point specialist. All of those properties would be desirable in a method to be used by developers of high-performance programs and large numerical simulations. This was the catalyst for our research.

My contribution to the analysis of floating-point computation is the development of a new method, *encapsulated error*, designed with high-performance computing and interpretability in mind. Using this method, we can measure the number of significant digits of any number and at any time in a computation which can be in mixed precision or computed in parallel. It seems important to stress that

our work is not applicable to all use cases, such as the formal verification of mathematical functions (an important use-case for floating-point analysis), but it is highly practical when one needs to deal with a large simulation as illustrated in Chapter 8. Furthermore, one can use this method as a building block to explore other avenues such as reduced precision, our original motivation.

As our method requires modifications to the source code, we quickly encountered scalability problems when wanting to evaluate programs too large to be manually instrumented. Our solution was the development of a code refactoring tool, based on the Clang compiler, able to change the types used in a program with minimal human involvement. While we are still limited to programs written in programming languages for which an implementation of encapsulated error has been provided, we can now instrument production programs with thousands of lines. This gave us the ability to test physical simulations that are currently used by physicists (as seen in Chapter 8), rather than toy problems. Doing so, working on problems of increasingly large size, we realized that numerical error cannot always be expected to grow linearly with problem size. Instead, we have observed sudden drops in number of significant digits (as seen in Section 8.3) while the loss of precision started small and progressive. This makes testing the accuracy of programs tricky, as small or medium test cases might not experiment this drop, leading the user to extrapolate and believe that his program will remain numerically stable for much larger problems.

Our final contribution is the introduction of a method to trace observed numerical errors to their sources. The need for such a tool came naturally once we found a numerically unstable program too large to be analyzed one operation at a time. Our solution was the development of *tagged error*, a variant of encapsulated error where we keep various user-defined sources of error separated from each other in order to be weight their individual contribution to the final result. Thus, a user can tag the coarse components of his program and get a report on their individual numerical stability and impact on further computations. An insight that came from tagged error, as seen in Chapter 8, is that numerical error does not always come from where we expect it. Intuition tells us that the most complex part of a program should be the at the source of the numerical error, but complex algorithms (such as the solving of linear systems or an eigenvalue decomposition) tend to rely on libraries and algorithms that have been numerically hardened with time. Meanwhile, conceptually simple initialization steps usually do not benefit from those years of refinements and, surprisingly, can become major sources of numerical error in the final output. Once detected, due to their algorithmic simplicity, those the easiest steps to improve the numerical accuracy of a program using, for example, compensated algorithms. Another interesting outcome of tagged error is the ability to do targeted numerical improvements, sacrificing a minimum of performance

for increased precision. A direct example would be our work on the conjugate gradient algorithm in Section 6.3. When using a linear solver, one can already use the residual as a measure of accuracy, but having the ability to programmatically locate sources of error lets us inject targeted compensated operations in order to trade some performance for additional precision. This can be done without having to redesign the algorithm or introduce additional elements such as a preconditioner.

All our work is open-source and we made both our reference implementation, the Shaman library, and our tools available online [Demeure and Chevalier, 2019, Demeure and Ancellin, 2020]. Shaman is currently used at CEA, to both validate simulation codes and evaluate the outputs of programs that have been compiled with a different compiler or on a new processor architecture. It is also featured in upcoming publications on uncertainty quantification methods and its usage is explored by researchers working on introducing mixed precision in computations. As our work currently stands, it would be interesting to invest engineering effort to produce highly performant implementations in common numerical languages – such as Python, Matlab and Fortran – coupled with solid instrumentation tools. A Valgrind-based implementation, in particular, would make the instrumentation of code bases relying on several programming languages much easier. From a research perspective, two directions stand out to me. First, tagged error would probably benefit greatly from algorithmic refinements and research on an efficient inner representation, similar to the work that has been done on affine arithmetic. Currently, if the user has defined ten tags, then all numbers will have ten errors terms even if it can be proven that most of those error terms are zero. Reducing waste in the representation could lead to large performance improvements. Second, I believe that one could use the possibility to access the numerical error *within the code* as a building block for new numerical algorithms and higher level tools. An example would be the use of our knowledge of the numerical error as part of the stopping criteria of an iterative algorithm.

Finally, I would like to stress that I purposefully focused on a single, simple, idea. While more complex solutions might be alluring, and while there is certainly pressure to focus on complex approaches, I strongly believe that simple ideas are both more resilient, having fewer points of failure, and much more likely to be re-used, being easy to implement and reproduce. Hopefully, having a simple solution that produces good accuracy estimations, is faster than existing approaches, and easy to use will help more people investigate the floating-point accuracy of their computations.

List of Figures

List of Tables

List of Code listings

List of Algorithms

List of published works

Papers:

- Demeure, N., Denis C., Chevalier C. and Dossantos-Uzarralde P., *Shaman: A direct approach to assess floating-point accuracy*, ACM Transactions on Mathematical Software (**under review**)

Talks:

- *Shaman : Une méthode directe et déterministe pour estimer l'erreur due à l'arithmétique flottante*, (2018), seminar of the Informatique et Calcul Parallèle Scientifique team of the Icube laboratory from the University of Strasbourg.
- *Shaman : Une méthode directe et déterministe pour estimer l'erreur due à l'arithmétique flottante*, (2018), CEA–EDF–INRIA summer school
- *Shaman: A direct and deterministic approach to access floating-point accuracy*, (2019), SimRace (**conference cancelled**)
- *Uncertainties and Numerical Accuracy*, (2020), Los Alamos / DPTA 2020 meeting
- *AI augmented linear solvers: using machine learning to predict the convergence profile of a linear solver*, (2020), Digital French-German Summer School with Industry (video available online)

Posters:

- Demeure, N., Denis C., Chevalier C. and Dossantos-Uzarralde P., *Uncertainty Quantification and Numerical Accuracy*, (2020), MASCOT-NUM
- Demeure, N., Denis C., Chevalier C. and Dossantos-Uzarralde P., *A Direct Method to Assess Floating-Point Accuracy*, (2021), Platform for Advanced Scientific Computing (PASC) Conference

Codes:

- Demeure, N. and Chevalier C. *Shaman C++ reference implementation*, (2019), Gitlab repository
- Demeure, N. and Ancellin M. *Shaman Julia reference implementation*, (2020), Gitlab repository

Bibliography

[Bailey, 1995] Bailey, D. H. (1995). A fortran 90-based multiprecision system. *ACM Transactions on Mathematical Software (TOMS)*, 21(4):379–387.

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

[Bao and Zhang, 2013] Bao, T. and Zhang, X. (2013). On-the-fly detection of instability problems in floating-point program execution. *SIGPLAN Not.*, 48(10):817–832.

[Barrett et al., 1996] Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and Van der Vorst, H. (1996). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM. Appendix D: Flowchart of iterative methods.

[Bavier et al., 2012] Bavier, E., Hoemmen, M., Rajamanickam, S., and Thornquist, H. (2012). Amesos2 and belos: Direct and iterative solvers for large sparse linear systems. *Scientific Programming*, 20(3):241–255.

[Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

[Benz et al., 2012] Benz, F., Hildebrandt, A., and Hack, S. (2012). A dynamic program analysis to find floating-point accuracy problems. *SIGPLAN Not.*, 47(6):453–462.

[Bersillon, 1988] Bersillon, O. (1988). Lectures on the computer code scat-2. In *Workshop on Applied Nuclear Theory and Nuclear Model Calculations for Nuclear Technology Applications,(Trieste. 1988)*.

[Bezanson et al., 2017] Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98.

[Bipm et al., 2008] Bipm, I., IFCC, I., and ISO, I. (2008). Iupap and oiml,"evaluation of measurement data–supplement 1 to the 'guide to the expression of uncertainty in measurement'–propagation of distributions using a monte carlo method". *Joint Committee for Guides in Metrology, JCGM*, 101.

[Boehm, 2020] Boehm, H.-J. (2020). Towards an api for the real numbers. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 562–576.

[Bohlender et al., 1991] Bohlender, G., Walter, W., Kornerup, P., and Matula, D. W. (1991). Semantics for exact floating point operations. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 22–26. IEEE.

[Brönnimann et al., 2006] Brönnimann, H., Melquiond, G., and Pion, S. (2006). The design of the boost interval arithmetic library. *Theoretical Computer Science*, 351(1):111–118.

[Chesneaux, 1988] Chesneaux, J.-M. (1988). Modélisation et conditions de validité de la méthode cestac. *Comptes rendus de l'Académie des sciences. Série 1, Mathématique*, 307(8):417–422.

[Chesneaux et al., 2009] Chesneaux, J.-M., Graillat, S., and Jézéquel, F. (2009). Numerical validation and assessment of numerical accuracy. `https://www-pequan.lip6.fr/~graillat/papers/oerc_numerical_accuracy.pdf`.

[Comba and Stolfi, 1993] Comba, J. L. D. and Stolfi, J. (1993). Affine arithmetic and its applications to computer graphics. In *7th Sibgrapi (Brazilian Symposium on Computer Graphics and Image Processing)*, pages 9–18.

[Dagum and Menon, 1998] Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55.

[Darulova and Kuncak, 2014a] Darulova, E. and Kuncak, V. (2014a). On numerical error propagation with sensitivity. Technical report, Ecole Polytechnique Federale de Lausanne.

[Darulova and Kuncak, 2014b] Darulova, E. and Kuncak, V. (2014b). Sound compilation of reals. *SIGPLAN Not.*, 49(1):235–248.

[Davis and Hu, 2011] Davis, T. A. and Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25.

[de Dinechin et al., 2011] de Dinechin, F., Lauter, C., and Melquiond, G. (2011). Certifying the floating-point implementation of an elementary function using gappa. *IEEE Transactions on Computers*.

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

[Demeure, 2020] Demeure, N. (2020). Ai augmented linear solvers: using machine learning to predict the convergence profile of a linear solver. `http://mlmda.cmla.fr/french-german-summer-school-for-industry-2020/`.

[Demeure and Ancellin, 2020] Demeure, N. and Ancellin, M. (2020). Shaman julia reference implementation. `https://gitlab.com/numerical_shaman/shaman_julia`.

[Demeure and Chevalier, 2019] Demeure, N. and Chevalier, C. (2019). Shaman c++ reference implementation. `https://gitlab.com/numerical_shaman/shaman`.

[Demmel et al., 2008] Demmel, J., Dumitriu, I., Holtz, O., and Koev, P. (2008). Accurate and efficient expression evaluation and linear algebra. *Acta Numerica*, 17:87–145.

[Denis et al., 2016] Denis, C., de Oliveira Castro, P., and Petit, E. (2016). Verificarlo: checking floating point accuracy through monte carlo arithmetic. In *23nd Symposium on Computer Arithmetic (ARITH)*, pages 55–62.

[Dossantos-Uzarralde and Guittet, 2008] Dossantos-Uzarralde, P. and Guittet, A. (2008). A polynomial chaos approach for nuclear data uncertainties evaluations. *Nuclear Data Sheets*, 109(12):2894–2899.

[Eça et al., 2019] Eça, L., Vaz, G., Toxopeus, S. L., and Hoekstra, M. (2019). Numerical Errors in Unsteady Flow Simulations. *Journal of Verification, Validation and Uncertainty Quantification*, 4(2).

[Forsythe et al., 1977] Forsythe, G. E., Malcolm, M. A., and Moler, C. B. (1977). *Computer Methods for Mathematical Computations*. Prentice Hall Professional Technical Reference.

[Fousse et al., 2007] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., and Zimmermann, P. (2007). Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2).

[Févotte and Lathuilière, 2016] Févotte, F. and Lathuilière, B. (2016). VERROU: a CESTAC evaluation without recompilation. In *International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics (SCAN)*, Uppsala, Sweden.

[Févotte and Lathuilière, 2017] Févotte, F. and Lathuilière, B. (2017). Verrou tutorial for the PRECIS summer school. `https://github.com/edf-hpc/verrou/tree/ecole-precis`. (in French).

[Giordano, 2016] Giordano, M. (2016). Uncertainty propagation with functionally correlated quantities. *arXiv preprint arXiv:1610.08716*.

[Glorot et al., 2011] Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323.

[Goubault, 2013] Goubault, E. (2013). Static analysis by abstract interpretation of numerical programs and systems, and fluctuat. In Logozzo, F. and Fähndrich, M., editors, *Static Analysis*. Springer Berlin Heidelberg.

[Gouy, 2020] Gouy, I. (2020). The computer language benchmarks game. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`. Last checked on Jan 10, 2020.

[Graillat et al., 2009] Graillat, S., Lamotte, J.-L., and Hong, D. N. (2009). Error-free transformation in rounding mode toward zero. In *Numerical Validation in Current Hardware Architectures*, pages 217–229. Springer.

[Gropp et al., 1996] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828.

[Guennebaud et al., 2010] Guennebaud, G., Jacob, B., et al. (2010). Eigen v3. `http://eigen.tuxfamily.org`.

[Guennebaud et al., 2018] Guennebaud, G., Jacob, B., et al. (2018). Eigen/reliability. `http://eigen.tuxfamily.org/index.php?title=Reliability`.

[Gustafson, 2015] Gustafson, J. L. (2015). *The End of Error: Unum Computing*. Chapman and Hall/CRC.

[Heroux et al., 2005] Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M.,

Williams, A., and Stanley, K. S. (2005). An overview of the trilinos project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423.

[Holoborodko, 2010] Holoborodko, P. (2010). Mpfr c++. `http://www.holoborodko.com/pavel/mpfr/`.

[IEEE, 2019] IEEE, C. M. S. C. (2019). IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*.

[Intel, 2013] Intel (2013). Intel fortran compiler / optimization question. `https://community.intel.com/t5/Intel-Fortran-Compiler/Optimization-question/td-p/930791`. Last checked on Sep 26, 2020.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

[Jensen et al., 2011] Jensen, F. B., Kuperman, W. A., Porter, M. B., and Schmidt, H. (2011). *Computational ocean acoustics*. Springer Science & Business Media.

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

[Johansson, 2017] Johansson, F. (2017). Arb: Efficient arbitrary-precision midpoint-radius interval arithmetic. *IEEE Transactions on Computers*, 66(8):1281 – 1292.

[Jézéquel, 2020] Jézéquel, F. (2020). Precision auto-tuning and control of accuracy in high performance simulations. `http://orap.irisa.fr/?page_id=1233`. ORAP, 45e Forum.

[Kahan, 1997] Kahan, W. (1997). Lecture notes on the status of ieee standard 754 for binary floating-point arithmetic. `https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF`.

[Kahan, 2004] Kahan, W. (2004). On the cost of floating-point computation without extra-precise arithmetic. `https://people.eecs.berkeley.edu/~wkahan/Qdrtcs.pdf`.

[Kahan, 2006] Kahan, W. (2006). How futile are mindless assessments of roundoff in floating-point computation? `http://www.cs.berkeley.edu/~wkahan/Mindless.pdf`.

[Kahan and Darcy, 1998] Kahan, W. and Darcy, J. D. (1998). How java's floating-point hurts everyone everywhere. In *ACM 1998 workshop on java for high-performance network computing*, page 81. Stanford University.

[Karlin, 2012] Karlin, I. (2012). Lulesh programming model and performance ports overview. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA.

[Khalaf et al., 2015] Khalaf, A., Khalifa, M. M., Solieman, A., and Comsan, M. (2015). Calculation of phase shift for p+40 ca elastic scattering at low energy. *Nature and Science*.

[Khalaf et al., 2018] Khalaf, A., Khalifa, M. M., Solieman, A., and Comsan, M. (2018). Nuclear matter parameters and optical model analysis of proton elastic scattering on the doubly magic nucleus 40ca. *Nuclear Physics A*, 969:83–93.

[Knizia et al., 2011] Knizia, G., Li, W., Simon, S., and Werner, H.-J. (2011). Determining the numerical stability of quantum chemistry algorithms. *Journal of chemical theory and computation*, 7(8):2387–2398.

[Knuth, 1998] Knuth, D. (1998). *The Art of Computer Programming vol. 2 Seminumerical Algorithms*. Reading, Massachusetts: Addison Wesley.

[Koning and Delaroche, 2002] Koning, A. and Delaroche, J. (2002). New neutron and proton optical models. Technical report, International Atomic Energy Agency (IAEA).

[Lange and Rump, 2020] Lange, M. and Rump, S. M. (2020). Faithfully rounded floating-point computations. *ACM Transactions on Mathematical Software (TOMS)*, 46(3):1–20.

[Latkin, 2014] Latkin, E. (2014). Twofold fast arithmetic. *arXiv preprint arXiv:1401.6235*.

[Lattner, 2008] Lattner, C. (2008). Llvm and clang: Next generation compiler technology. In *The BSD conference*, volume 5.

[Le Maıtre et al., 2004] Le Maıtre, O., Najm, H. N., Ghanem, R., and Knio, O. (2004). Multi-resolution analysis of wiener-type uncertainty propagation schemes. *Journal of Computational Physics*, 197(2):502–531.

[Loshchilov and Hutter, 2017] Loshchilov, I. and Hutter, F. (2017). Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

[Luk et al., 2005] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200.

[Marques et al., 2017] Marques, D., Duarte, H., Ilic, A., Sousa, L., Belenov, R., Thierry, P., and Matveev, Z. A. (2017). Performance analysis with cache-aware roofline model in intel advisor. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pages 898–907. IEEE.

[McCulloch and Pitts, 1943] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

[Moore, 1963] Moore, R. E. (1963). *Interval arithmetic and automatic error analysis in digital computing*. Stanford University.

[Moore, 1966] Moore, R. E. (1966). Interval analysis. *Prentice-Hall, Englewood Cliffs, NJ*.

[Muller, 1989] Muller, J.-M. (1989). *Arithmétique des ordinateurs*. Masson.

[Muller, 2005] Muller, J.-M. (2005). On the definition of ulp(x). Technical Report RR-5504, LIP RR-2005-09, INRIA, LIP.

[Muller et al., 2010] Muller, J.-M., Brisebarre, N., De Dinechin, F., Jeannerod, C.-P., Lefevre, V., Melquiond, G., Revol, N., Stehlé, D., and Torres, S. (2010). *Handbook of floating-point arithmetic*. Birkhäuser Boston.

[Nethercote and Seward, 2007] Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100.

[Panchekha et al., 2015] Panchekha, P., Sanchez-Stern, A., Wilcox, J. R., and Tatlock, Z. (2015). Automatically improving accuracy for floating point expressions. *SIGPLAN Not.*, 50(6):1–11.

[Parker, 1997] Parker, D. S. (1997). Monte carlo arithmetic: exploiting randomness in floating-point arithmetic. Technical report, University of California (Los Angeles). Computer Science Department.

[Priest, 1992] Priest, D. M. (1992). *On properties of floating point arithmetics: numerical stability and the cost of accurate computations*. PhD thesis, University of California, Berkeley.

[Prokopenko et al., 2016] Prokopenko, A., Siefert, C. M., Hu, J. J., Hoemmen, M., and Klinvex, A. (2016). Ifpack2 User's Guide 1.0. Technical Report SAND2016-5338, Sandia National Labs.

[Rahman, 2012] Rahman, A. K. M. R. (2012). Neutron cross-sections for^{55} mn in the energy range from 0.2 to 22 mev. *Turkish Journal of Physics*, 36(3):343–351.

[Regnier et al., 2017] Regnier, D., Dubray, N., Verrière, M., and Schunck, N. (2017). Felix-2.0: New version of the finite element solver for the time dependent generator coordinate method with the gaussian overlap approximation. *Computer Physics Communications*.

[Regnier et al., 2016] Regnier, D., Verriere, M., Dubray, N., and Schunck, N. (2016). Felix-1.0: A finite element solver for the time dependent generator coordinate method with the gaussian overlap approximation. *Computer Physics Communications*, 200:350–363.

[Revels et al., 2016] Revels, J., Lubin, M., and Papamarkou, T. (2016). Forward-mode automatic differentiation in Julia. *arXiv preprint arXiv:1607.07892*.

[Revol, 2003] Revol, N. (2003). Interval newton iteration in multiple precision for the univariate case. *Numerical Algorithms*, 34(2-4):417–426.

[Rubio-González et al., 2013] Rubio-González, C., Nguyen, C., Nguyen, H. D., Demmel, J., Kahan, W., Sen, K., Bailey, D. H., Iancu, C., and Hough, D. (2013). Precimonious: Tuning assistant for floating-point precision. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12.

[Rump, 1983] Rump, S. M. (1983). Wie zuverlässig sind die ergebnisse unserer rechenanlagen? (how reliable are results of computers?). *Jahrbuch Uberblicke Mathematik*.

[Rump, 1988] Rump, S. M. (1988). Algorithms for verified inclusions: Theory and practice. In *Reliability in computing*, pages 109–126. Elsevier.

[Schatz, 2014] Schatz, V. (2014). fltscale. `https://www.volkerschatz.com/science/pics/fltscale.svg`. Last checked on Aug 21, 2020.

[Severance, 1998] Severance, C. (1998). An interview with the old man of floating point. *IEEE Computer*, 20(1):114–115.

[Slater, 2008] Slater, J. (2008). Uncertainty and error in cfd simulations. *NPARC Verfication and Validation Web site Home Page*.

[Smith, 2017] Smith, L. N. (2017). Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE.

[Solovyev et al., 2015] Solovyev, A., Jacobsen, C., Rakamarić, Z., and Gopalakrishnan, G. (2015). Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In Bjørner, N. and de Boer, F., editors, *FM 2015: Formal Methods*. Springer International Publishing.

[Stallman et al., 2002] Stallman, R., Pesch, R., Shebs, S., et al. (2002). *Debugging with GDB*. Free Software Foundation, Inc.

[Stannered, 2008] Stannered (2008). Float example. `https://commons.wikimedia.org/wiki/File:Float_example.svg`. Last checked on Aug 21, 2020.

[Sterbenz, 1974] Sterbenz, P. H. (1974). *Floating-point computation*. Prentice-Hall.

[Strzeboński, 1997] Strzeboński, A. W. (1997). Computing in the field of complex algebraic numbers. *Journal of Symbolic Computation*, 24(6):647–656.

[Sun, 1979] Sun, T.-C. (1979). A finite element method for random differential equations with random coefficients. *SIAM Journal on Numerical Analysis*, 16(6):1019–1035.

[Thévenoux et al., 2015] Thévenoux, L., Langlois, P., and Martel, M. (2015). Automatic source-to-source error compensation of floating-point programs. In *2015 IEEE 18th International Conference on Computational Science and Engineering*, pages 9–16. IEEE.

[Vignes, 1984] Vignes, J. (1984). Implémentation des méthodes d'optimisation : test d'arrêt optimal, contrôle et précision de la solution. *RAIRO - Operations Research - Recherche Opérationnelle*, 18(1):1–18.

[Vignes, 2004] Vignes, J. (2004). Discrete Stochastic Arithmetic for Validating Results of Numerical Software. *Numerical Algorithms*, 37(1-4):377–390.

[Vignes and ARSAC, 1986] Vignes, J. and ARSAC, J. (1986). Zéro mathématique et zéro informatique. *Comptes rendus de l'Académie des sciences. Série 1, Mathématique*, 303(20):997–1000.

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

[Waxler et al., 2017] Waxler, R., Assink, J., and Velea, D. (2017). Modal expansions for infrasound propagation and their implications for ground-to-ground propagation. *The Journal of the Acoustical Society of America*, 141(2):1290–1307.

[Whitehead and Fit-Florea, 2011] Whitehead, N. and Fit-Florea, A. (2011). Precision & performance: Floating point and ieee 754 compliance for nvidia gpus. `https://developer.nvidia.com/content/precision-performance-floating-point-and-ieee-754-compliance-nvidia-gpus`.

[Wiener, 1938] Wiener, N. (1938). The homogeneous chaos. *American Journal of Mathematics*, 60(4):897–936.

[Wilkinson, 1964] Wilkinson, J. H. (1964). *Rounding errors in algebraic processes*. Courier Corporation.

[Williams and Rasmussen, 2006] Williams, C. K. and Rasmussen, C. E. (2006). *Gaussian processes for machine learning*. MIT press Cambridge, MA.

[Yang et al., 2018] Yang, C., Gayatri, R., Kurth, T., Basu, P., Ronaghi, Z., Adetokunbo, A., Friesen, B., Cook, B., Doerfler, D., Oliker, L., Deslippe, J., and Williams, S. (2018). An empirical roofline methodology for quantitatively assessing performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 14–23.

[Yeo, 2020] Yeo, D. (2020). A summary of industrial verification, validation, and uncertainty quantification procedures in computational fluid dynamics. Technical report, NIST.

[Yeomn et al., 2016] Yeomn, J.-S., Thiagarajan, J. J., Bhatele, A., Bronevetsky, G., and Kolev, T. (2016). Data-driven performance modeling of linear solvers for sparse matrices. In *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 32–42. IEEE.

[Zeller, 2009] Zeller, A. (2009). *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition.

[Zikanov, 2019] Zikanov, O. (2019). *Essential computational fluid dynamics*. John Wiley & Sons.

[Zou et al., 2019] Zou, D., Zeng, M., Xiong, Y., Fu, Z., Zhang, L., and Su, Z. (2019). Detecting floating-point errors via atomic conditions. *Proc. ACM Program. Lang.*, 4.

Résumé étendu à l'intention du lecteur francophone

Introduction

Ce qui suit est un résumé étendu de la thèse. Il est, du fait de sa taille réduite, nécessairement moins complet que le manuscrit anglophone. Néanmoins, j'ai préservé l'introduction, la conclusion et les contributions clefs des différentes parties pour le lecteur francophone intéressé.

Alors que la loi de Moore atteint ses limites, les auteurs de programmes informatiques ne peuvent plus compter sur l'arrivée de nouveaux processeurs pour améliorer significativement leurs performances. De ce fait, ils se concentrent maintenant sur de nouvelles voies telles que l'introduction d'architectures différentes comme les processeurs graphiques (GPU), le parallélisme, pour obtenir plus de puissance en ajoutant plus de processeurs à la tâche, et les algorithmes en précision mixte pour améliorer le nombre d'opérations par seconde en réduisant la précision employée localement. Toutes ces approches (et, pourrait-on dire, l'écriture d'algorithmes numériques dans un contexte de calculs haute performance en général) nécessitent un contrôle sur le compromis entre performance et précision. En effet, pour passer à une précision mixte, il est nécessaire de savoir déterminer quelles parties du programme peuvent encore produire des résultats précis lorsqu'elles sont calculées avec une précision réduite. Le lien entre performance et précision apparaît également avec l'utilisation d'algorithmes numériques parallèles. Ceux-ci rendent explicite la non-associativité de l'arithmétique en virgule flottante en produisant régulièrement des résultats légèrement différents d'une exécution à l'autre, ce qui amène les utilisateurs à s'interoger sur la reproductibilité de leur résultats. Enfin, la plupart des GPU ne peuvent atteindre leurs performances maximales qu'avec une précision de 32 bit alors que la majorité des logiciels scientifiques actuels sont développés en arithmétique à virgule flottante 64 bit.

Pour résoudre ces problèmes, il est nécessaire de s'intéresser à la précision des résultats. Nous nous demandons donc comment calculer cette précision. L'objectif de ce doctorat est de donner des outils aux développeurs afin qu'ils puissent prendre des décisions éclairées au moment d'échanger de la précision contre des performances. Nous mettons l'accent sur la simulation et le calcul haute performance. Bien que nous concentrions nos efforts sur la précision numérique, nous détaillerons également une approche pour la sélection d'un solveur linéaire et d'un préconditionneur dans la partie IV.

Notre question de recherche initiale était de déterminer s'il est possible d'augmenter la performance d'une application en diminuant la précision localement, sans compromettre l'exactitude de ses résultats. Cependant, nous avons rapidement fait

face à un manque de méthodes viables pour mesurer la précision numérique sur les problèmes auxquels nous nous intéressons. Des recherches plus approfondies ont révélé que la vérification systématique de la précision numérique d'une application n'est pas une pratique courante. Cela signifie que, souvent, seuls les problèmes numériques les plus visibles sont susceptibles d'être détectés.

Nous avons donc décidé de nous concentrer sur la mesure de l'erreur numérique et, en particulier, sur son application à la simulation et au calcul haute performance. Il s'agit d'un domaine où les applications sont de grande taille, où elles peuvent avoir des durées d'exécution longues et un nombre élevé d'opérations par seconde. Par conséquent, c'est un domaine qui nécessite une méthode capable de passer à l'échelle.

Comme nous le détaillons dans le chapitre 3, l'évaluation de la précision numérique d'une application est un problème difficile. Bien que de bonnes solutions aient été développées, elles ne conviennent souvent pas aux très grande applications en raison de leurs hypothèses de travail ou sont très lentes et donnent des résultats difficiles à interpréter. Le problème de l'identification des sources d'erreur numérique dans une application inexacte souffre des mêmes lacunes.

Notre principale contribution est le développement et la validation d'une nouvelle méthode, que nous appelons *erreur encapsulée*, pour mesurer l'erreur numérique d'une application et localiser ses sources d'erreur. Pour résumer, nous calculons l'erreur numérique produite localement et la propageons dans des calculs ultérieurs via un type dédié qui encapsule à la fois le résultat du calcul initial et une approximation de son erreur numérique. Cette méthode nous donne un accès direct au nombre de chiffres significatifs à tout moment de l'exécution du programme et pour tous les résultats intermédiaires. Même si elle est simple, cette méthode est compétitive avec l'état de l'art, compatible avec diverses formes de parallélisme et a un faible surcoût, ce qui la rend adaptée au calcul haute performance et à la simulation, comme l'illustrent diverses applications dans le domaine de la simulation physique (chapitre 8). En outre, elle se compose d'opérations qui sont disponibles dans la plupart des langages de programmation, ce qui la rend facile à mettre en œuvre (nous fournissons des implémentations de référence, sous le nom de *Shaman* et *Shaman_julia*, dans les langages de programmation C++ et Julia), et donne des résultats facilement interprétables qui peuvent être analysés par un non-expert. C'est une propriété particulièrement importante car la plupart des codes numériques ne sont pas écrits par des analystes numériques. Nous espérons que cette méthode rendra l'analyse de la précision numérique d'une application à la fois plus accessible et plus viable pour un large éventail d'applications.

Ce résumé est divisé en quatre sections qui reflètent les contributions de cette thèse. La première partie résume brièvement l'état de l'art en matière de mesure de l'erreur numérique et de localisation des sources d'erreur dans une application.

La deuxième partie introduit notre principale contribution : notre méthode pour mesurer et localiser les sources d'erreur numérique. La troisième partie montre à la fois la précision de la méthode et son coût d'exécution réduit, par rapport à l'état de l'art, sur une série de cas de tests. La dernière partie ne traite pas de l'erreur numérique mais revient sur le compromis entre précision et performance. Elle explore une utilisation intéressante de l'apprentissage statistique : prédire le profil de convergence d'un solveur linéaire pour un nouveau système linéaire. Et ce, afin de sélectionner des solveurs et des préconditionneurs plus rapides et plus précis pour améliorer les résultats tout en continuant à utiliser un solveur déjà éprouvé.

10.1 État de l'art

10.1.1 Mesurer l'erreur numérique

Notre objectif est le développement d'une méthode pour quantifier l'impact de l'erreur numérique sur les sorties d'une application donnée dans le cadre de la simulation et du calcul haute performance.

L'approche la plus simple et directe pour résoudre ce problème consiste à comparer le résultat avec un résultat calculé en plus haute précision. Elle est, par exemple, implémentée dans FpDebug [Benz et al., 2012] et Precimonious [Rubio-González et al., 2013]. Le fait de ne pas pouvoir déterminer à l'avance si la précision employée sera suffisante est une limite importante de cette méthode. En effet, une *cancelation* suffisamment large peut impacter de la même manière un calcul en 32 bit et un calcul en 200 bit.

Une autre approche consiste à faire une analyse statique du code pour borner l'erreur de ses sorties. Elle est, par exemple, implémentée par FLUCTUAT [Goubault, 2013] et Rosa [Darulova and Kuncak, 2014b]. Cependant, à l'heure actuelle, cette approche ne s'applique qu'aux programmes relativement simples (fonction unique, pas de boucle aux conditions d'arret non triviales).

L'arithmétique des intervalles [Moore, 1963, Moore, 1966] permet de s'assurer des mêmes garanties fortes tout en palliant ces problèmes. Cette méthode consiste à propager les bornes de l'intervalle lors de l'exécution du programme. Malheureusement, de nombreux programmes pourtant valides produisent des intervalles trop larges pour être exploitables.

L'arithmétique stochastique [Vignes and La Porte, 1974, Parker, 1997] correspond à une approche transversale du problème. Pour l'implémenter, on injecte du bruit lors des opérations et on fait tourner le programme plusieurs fois pour observer la distribution de ses sorties. Le fait de devoir produire un nombre aléatoire par opération et de faire tourner le code plusieurs fois est une limite de cette approche, le programme s'en trouvant considérablement ralenti.

Finalement, il est possible de mesurer l'erreur produite localement, au niveau des opérations individuelles. Celle-ci peut donner une idée de la qualité d'une application. Néanmoins, cette approche est sujette aux faux positifs et faux négatifs car elle ne permet pas de mesurer l'impact d'une erreur locale sur la sortie du programme.

Dans ces conditions, il nous semble important de développer notre propre méthode, qui soit à la fois fiable et viable pour l'analyse de gros programmes de simulation.

10.1.2 Localiser les sources d'erreur numérique

Une fois que nous avons déterminé qu'un programme souffre d'erreurs numériques, l'étape suivante consiste à identifier les sources de ces erreurs. Les solutions à ce problème sont rares et se regroupent en deux catégories.

D'un côté, nous pouvons utiliser une détection locale et signaler les opérations qui produisent la plus grosse erreur numérique (typiquement des *cancelations*) en supposant que cette erreur aura un impact proportionnel sur la sortie finale. Cette approche, qui est représentée par le debugger numérique de CADNA [Jézéquel and Chesneaux, 2008] par exemple, souffre des problèmes de faux positifs et faux négatifs des approches locales.

D'un autre côté, nous pouvons désactiver la localisation d'erreur dans une section du code et le faire tourner plusieurs fois en faisant varier les paramètres de la section du code suivant un plan d'expérience (typiquement via un algorithme de delta-debugging [Zeller, 2009]) pour en déduire la contribution des diverses sources d'erreur potentielles. Cette méthode, que nous trouvons dans Precimonius [Rubio-González et al., 2013] et Verrou [Févotte and Lathuilière, 2016] par exemple, requiert un grand nombre d'évaluations. Ceci la rend particulièrement lente et peut renvoyer une information peu claire sur la localisation de la source d'erreur.

Dans les sections qui suivent, nous proposons une méthode qui se veut plus rapide et simple d'utilisation.

10.2 L'erreur encapsulée, une nouvelle méthode

10.2.1 Erreur encapsulée

Nous proposons l'utilisation de ce que nous appelons l'*erreur encapsulée* pour mesurer l'erreur numérique d'une application. Notre idée consiste à remplacer les nombres à virgule flottante par des paires $(number, error)$ qui contiennent à la fois le résultat du calcul original en arithmétique à virgule flottante ($number$) et une approximation de son erreur numérique ($error$).

Le calcul du résultat d'une opération arithmétique entre deux nombres ainsi instrumentés passe par le calcul de l'erreur produite par l'opération arithmétique à l'aide d'une opération appelée *error-free transformation* [Muller et al., 2010] et la propagation de l'erreur issue des arguments dans l'erreur de la sortie. Les opérateurs arithmétiques sont définis comme suit :

Algorithm 15 Addition((x, δ_x), (y, δ_y))

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

Algorithm 16 Multiplication((x, δ_x), (y, δ_y))

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

Algorithm 17 Division((x, δ_x), (y, δ_y))

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

En l'absence d'*error-free transformation* (pour les fonctions quelconques comme le cosinus), nous déduisons l'erreur finale en soustrayant le résultat de l'opération à un résultat calculé en arithmétique haute précision.

Les comparaisons sont faites en utilisant uniquement la partie *number* des paires et seuls les chiffres du résultat qui sont significatifs étant donnée l'erreur numérique calculée, sont affichés.

10.2.2 Erreur taguée

Nous proposons une extension de la méthode, que nous appelons *erreur taguée*, pour la localisation des sources d'erreur. Cette méthode consiste à déterminer un terme d'erreur par section du code que l'utilisateur délimite. Les termes d'erreur restent séparés, indépendants les uns des autres et permettent de déterminer la section (identifiée par son nom aussi appelé *tag*) d'où provient une erreur numérique donnée.

L'addition est alors définie comme suit :

Algorithm 18 Addition($(x, [\delta_{xi}])$, $(y, [\delta_{yi}])$)

$z \leftarrow x + y$
for i **in** tags **do**
  $\epsilon_i \leftarrow$ **if** CurrentTag() $== i$ **then** TwoSum(x, y) **else** 0.0
  $\delta_{zi} \leftarrow \delta_{xi} + \delta_{yi} + \epsilon_i$
return $(z, [\delta_{zi}])$

L'affichage des nombres est aussi modifié afin d'afficher non seulement un nombre avec ses chiffres significatifs mais aussi la provenance des termes d'erreur en pourcentages. Par exemple :

$$1.23\ [func1 : 90\%, func2 : -20\%, ...]$$

10.2.3 Utilisation

Nous avons développé des implémentations de référence en C++ sous le nom Shaman [Demeure and Chevalier, 2019] et en Julia sous le nom Shaman_Julia [Demeure and Ancellin, 2020]. Le code 10.1 est un exemple de code C++ qui calcule la racine carrée avec l'algorithme de Héron, instrumenté en utilisant Shaman. La seule modification imposée par l'utilisation de Shaman est le remplacement des types *double* par des *Sdouble*. Ceci donne la possibilité à l'utilisateur d'accéder aux nombres et à leur erreur numérique n'importe où dans le programme. Ce remplacement peut être fait manuellement ou à l'aide d'un outil que nous avons développé à partir du compilateur Clang [Lattner, 2008].

```
1  // Sdouble definition:
2  // using Sdouble = S<double double, long double>;
3
4  Sdouble heron(Sdouble x)
5  {
6      Sdouble r = x/2;
7      int i = 0;
8
9      while(1e-15 < abs(r*r - x))
10     {
11        r = (r + x/r) / 2;
12        i++;
13        std::cout << "iteration:" << i << '␣'
14                  // displays only significant digits
```

```
15                  << "sqrt:" << r << std::endl
16                  // direct access to the number
17                  << "number:" << r.number << std::endl
18                  // direct access to the error
19                  << "error:" << r.error << std::endl
20                  << std::endl;
21      }
22
23      return r;
24  }
25
26  void main()
27  {
28      heron(2);
29  }
```

Code listing 10.1: instrumentation d'un code avec Shaman.

Nous affichons la sortie du code en activant le débugueur numérique pour obtenir un résumé des instabilités, nous obtenons alors le code 10.2.

```
iteration:1 sqrt:1.5000000000000000e+00
number:1.500000000000000e+00
error:-0.000000000000000e+00

iteration:2 sqrt:1.41666666666667e+00
number:1.416666666666667e+00
error:1.480297366166875e-16

iteration:3 sqrt:1.414215686274510e+00
number:1.414215686274510e+00
error:1.393221050510000e-16

iteration:4 sqrt:1.414213562374690e+00
number:1.414213562374690e+00
error:4.079910214529664e-17

iteration:5 sqrt:1.414213562373095e+00
number:1.414213562373095e+00
error:1.253716726897950e-16

*** SHAMAN ***
```

```
There are 5 numerical instabilities
3 CANCELLATION(S)
0 UNSTABLE DIVISION(S)
0 UNSTABLE MULTIPLICATION(S)
0 UNSTABLE MATHEMATICAL FUNCTION(S)
0 UNSTABLE POWER FUNCTION(S)
2 UNSTABLE BRANCHING(S)
```

Code listing 10.2: sorties du code quand le débugueur numérique est activé.

Les *cancelations* et branches instables peuvent être localisées avec des *break points* ou avec le profileur numérique qui nous indique, dans le code 10.3, que les opérations problématiques sont situées dans la condition d'arrêt de la boucle (la ligne 62 correspond à la condition d'arrêt de la boucle située dans la fonction *heron* du fichier *main.cpp*).

```
*** SHAMAN PROFILE ***
5       heron (file main.cpp)
3               operator- (line 62)
1               operator< (line 62)
1               abs (line 62)
```

Code listing 10.3: sorties du profileur numérique.

Cet exemple illustre le fait que, moyennant des modifications minimales (un changement de type), notre méthode nous donne accès à l'erreur numérique avec une granularité très fine.

10.3 Évaluation de la méthode

10.3.1 Précision

Pour évaluer la précision de notre méthode, nous avons intégré la fonction cosinus entre 0 et $\frac{\pi}{2}$ en utilisant la méthode des rectangles (ce cas test a été pensé par les auteurs de Verrou [Févotte and Lathuilière, 2016] et publié dans [Févotte and Lathuilière, 2017]). La valeur de cette intégrale est connue analytiquement (1) et nous savons que les deux seules sources d'erreur sont l'erreur numérique liée aux calculs en précision finie (ici 32 bit) et l'erreur de discrétisation qui, elle, décroit avec le nombre de rectangles. Ainsi, nous savons que pour un grand nombre de rectangles, l'erreur observée en comparant le résultat obtenu avec le résultat attendu analytiquement correspond à l'erreur numérique.

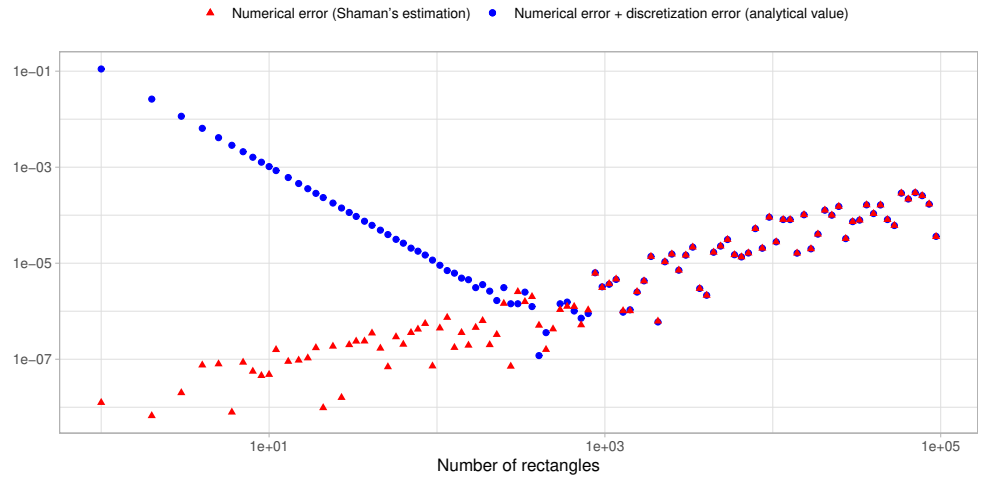


Figure 10.1: valeur absolue de l'erreur en fonction du nombre de rectangles utilisés pour l'intégration du cosinus entre 0 et $\frac{\pi}{2}$ en utilisant la méthode des rectangles. Les deux axes sont représentés en échelle logarithmique.

Lorsque nous traçons la courbe correspondant à la différence entre le résultat de l'intégration et la valeur analytique en fonction du nombre de rectangles (en bleu sur la figure 10.1), nous observons que l'erreur totale décroît au début (ce qui est attendu car l'erreur de discrétisation diminue) mais augmente de nouveau à partir d'un certain nombre de rectangles. Pour expliquer ce comportement, nous affichons l'estimation de l'erreur numérique en fonction du nombre de rectangles (en rouge sur la figure 10.1). Nous remarquons alors que l'erreur numérique augmente avec le nombre de rectangles et finit par devenir la source d'erreur dominante. Nous observons également que l'estimation de l'erreur numérique recouvre parfaitement l'erreur calculée analytiquement, montrant ainsi la précision de notre méthode.

10.3.2 Surcoût en temps de calcul

Un de nos objectifs était le développement d'une méthode suffisamment rapide pour pouvoir analyser de grandes applications. Cette section illustre le temps de calcul de Shaman, comparé à celui de l'état de l'art des méthodes de mesure de l'erreur numérique, sur un panel de cas tests (Lulesh 1.0 [Karlin, 2012] et les cas tests denses en nombre d'opérations arithmétiques du Computer Benchmark Game [Gouy, 2020]). Nous instrumentons les cas tests suivants :

- *n-body* : simulation à N corps ;
- *spectral norm* : calcul d'une valeur propre par la méthode de la puissance itérée ;

- *Mandelbrot set* : génération de l'ensemble de Mandelbrot avec une précision donnée ;
- *Lulesh 1.0* : résolution des équations d'hydrodynamique explicites sur une collection d'éléments volumétriques.

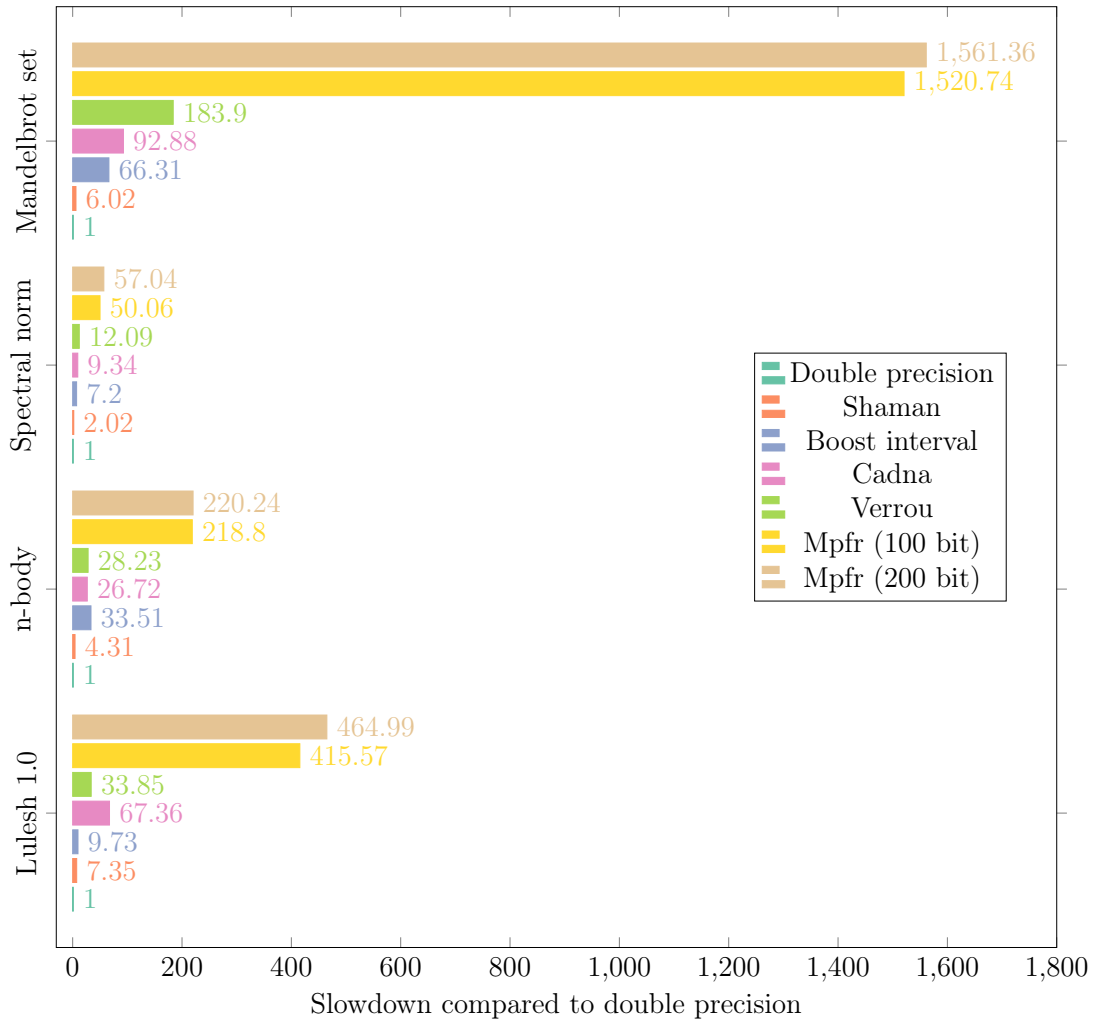


Figure 10.2: surcoût en temps de calcul de la librairie Shaman comparé à ceux de l'état de l'art.

La Figure 10.2 illustre les surcoûts en temps de calcul. On peut y observer que notre méthode, Shaman, a le surcoût le plus faible sur toutes les applications tests. En particulier, le surcoût observé pour Lulesh 1.0 (l'application la plus proche d'une simulation physique réaliste) est cohérent avec ce que nous avons observé

lors de nos tests sur de plus larges applications : les applications instrumentées avec Shaman tournent six à sept fois plus lentement, un surcoût sensiblement plus faible que ceux observés dans l'état de l'art.

Nos tests ont montré que notre implémentation de l'erreur taguée rend un code environ 50 fois plus lent pour l'utilisation de 10 tags (le temps de calcul augmente linéairement avec le nombre de tags). Le surcoût peut paraître conséquent mais cette extension de la méthode est conçue pour être employée uniquement quand un problème a déjà été détecté et que l'utilisateur veut remontrer aux sources de l'erreur numérique. Cette fonctionnalité est sensiblement plus complexe que la détection de la présence d'une erreur numérique dans les calculs.

10.4 Prédiction du profil de convergence d'un solveur linéaire

Les systèmes linéaires apparaissent dans toutes les formes de simulation mais la sélection d'un solveur et d'un préconditionneur adapté à un nouveau problème linéaire se fait encore, essentiellement, par tâtonnement. Cette façon de faire conduit les utilisateurs à se limiter dans le choix du solveur et du préconditionneur par excès de prudence et peut rendre la vitesse de convergence sous optimale.

Dans cette section, qui se focalise sur le compromis entre précision et performance plutôt que sur l'erreur numérique en particulier, nous proposons une application de l'apprentissage statistique (*machine learning*) pour résoudre ce problème. Une particularité de cette application est de ne pas proposer de remplacer les solveurs. L'apprentissage statistique est mis en oeuvre pour aider à la décision mais la résolution du système linéaire continue de se faire avec un solveur prouvé et aux propriétés connues.

10.4.1 Définition du problème

Le choix de la bonne paire solveur/préconditionneur pour un système linéaire permet d'obtenir des résultats précis et rapides mais il s'agit d'un problème combinatoire et multi-objectifs. Dans notre travail, nous couvrons onze solveurs itératifs de type Lanczos (issus de la bibliothèque Belos [Bavier et al., 2012]) et huit préconditionneurs (issus de la bibliothèque IfPack2 [Prokopenko et al., 2016]). En utilisant uniquement ces solveurs linéaires et, optionnellement, un préconditionneur droit ou gauche, il y a déjà $11 * (1 + 2 * 8) = 187$ combinaisons possibles.

Un utilisateur typique ne dispose pas d'un temps de calcul infini et a besoin d'atteindre une certaine précision dans ses résultats, dans le temps imparti. Pour l'aider à sélectionner un solveur et un préconditionneur en prenant ces deux

objectifs en compte, nous proposons un modèle qui prédit le profil de convergence des différents solveurs utilisables pour un problème donné.

Un profil de convergence est une courbe qui prend le temps de calcul en abscisse et le résidu relatif en ordonnée. Sur la figure 10.3, par exemple, nous pouvons observer que le choix du solveur le plus précis change en fonction du temps de calcul disponible.

Notre objectif est de prédire un profil de convergence avec suffisamment de précision pour pouvoir aider un utilisateur à prendre une décision éclairée.

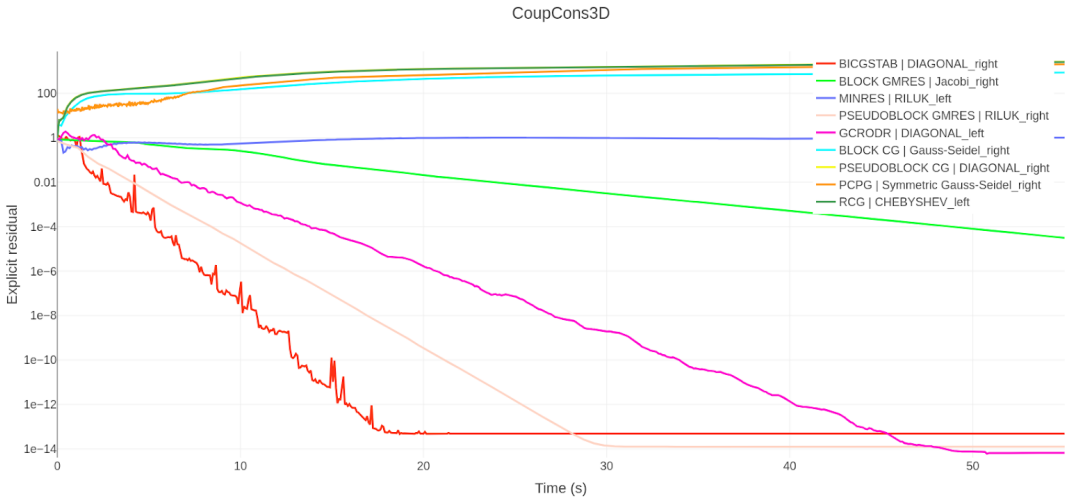


Figure 10.3: profil de convergence d'un échantillon aléatoire de solveurs linéaires et préconditionneurs pour le problème CoupCons3D. Le résidu relatif est affiché en échelle logarithmique.

10.4.2 Notre modèle

Notre solution est basée sur des réseaux de neurones artificiels [McCulloch and Pitts, 1943]. Nous prédisons le résidu des 2000 premières itérations d'une paire solveur/préconditionneur en fonction du temps de calcul. Le nombre d'itérations étant fixé, les courbes prédites sont donc plus ou moins longues en fonction du temps de calcul nécessaire pour effectuer une itération.

Pour réaliser cette prédiction, nous proposons d'utiliser deux réseaux dont les sorties sont combinées. Les deux modèles prennent des paramètres décrivant le système linéaire à résoudre, le solveur et le préconditionneur en entrée. Le premier modèle a 2000 sorties et prédit le résidu relatif pour les 2000 premières itérations. Le second modèle produit le logarithme du temps de calcul moyen par itération, ce qui permet de tracer la courbe correspondant à la sortie du premier modèle en

fonction du temps de calcul, reconstituant le profil de convergence.

Le modèle a été entraîné sur 1500 matrices carrées réelles (utilisant 20% des matrices comme ensemble de validation) avec l'optimiseur AdamW [Loshchilov and Hutter, 2017]. Nous avons utilisé l'erreur quadratique moyenne (*root mean square error*) comme mesure d'erreur pour l'apprentissage du temps de calcul et l'erreur absolue moyenne (*mean absolute error*) pour l'apprentissage des courbes de résidu. Les hyperparamètres de l'optimiseur (taux d'apprentissage, régularisation L2...) ont été sélectionnés par optimisation bayésienne.

10.4.3 Résultats

Une fois entraîné, le modèle atteint une erreur quadratique moyenne de **0.4** pour la prédiction du logarithme du temps de calcul et une erreur absolue moyenne de **4.5** pour la prédiction de la courbe résiduelle.

La figure 10.4 illustre une sortie typique. Les courbes de prédiction sont affichées en pointillés et les courbes qui correspondent à ce qui se passe en réalité en faisant le choix de ces solveurs sont en traits continus.

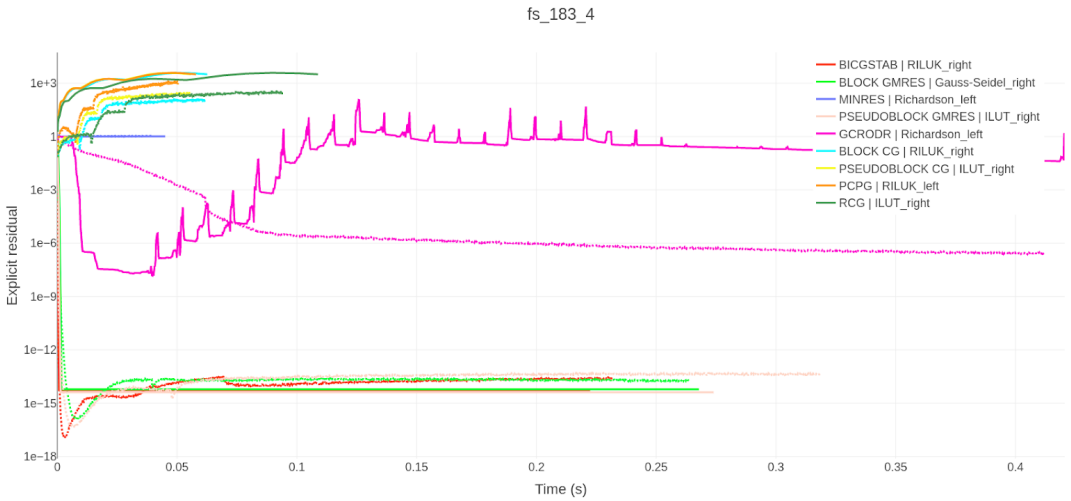


Figure 10.4: profil de convergence d'un échantillon aléatoire de solveurs linéaires et préconditionneurs pour le problème fs_183_4.

La prédiction n'est *pas* parfaite (ce que nous voyons bien sur la courbe correspondant au solveur GCRODR, tracée en magenta) mais elle fournit des informations suffisamment riches pour prendre une décision sur le choix du solveur à utiliser. En particulier, nous observons trois groupes dans la prédiction : les solveurs qui ne convergent pas (avec un résidu relatif supérieur à 1), un solveur qui converge lentement et vers une valeur peu précise (GCRODR, la divergence n'est pas observée

dans la prédiction mais le résidu final est correct) et des solveurs qui convergent rapidement et vers un résidu faible. De plus, en comparant les courbes de prédiction et les courbes correspondant à la réalité, nous observons que les groupes repérés ici sont associés aux bons solveurs et les résidus minimums sont correctement prédits.

10.4.4 Synthèse

Notre solution permet d'extraire des informations suffisamment précises pour sélectionner un solveur linéaire et un préconditionneur approprié pour résoudre un système linéaire donné, permettant d'améliorer à la fois le temps de calcul et la précision du résultat. C'est une problématique intéressante, du fait de ses gains potentiels, et qui nous semble encore peu explorée.

Cette expérience nous a permis de constater l'importance des données. En effet, lors de nos tests, augmenter la taille de l'ensemble d'entraînement a toujours conduit à une amélioration du modèle, significative et sensiblement plus grande que les améliorations dues à l'introduction de modèles plus complexes ou d'une architecture plus maligne.

Finalement, il semble important d'insister sur le fait que nous cherchons à compléter des algorithmes existants et *non* à les remplacer. Cela signifie que, tout en améliorant les temps de calcul et la précision finale, le système est toujours résolu par un solveur linéaire classique dont on a déjà prouvé qu'il fonctionne. Ce concept, augmenter ou compléter plutôt que remplacer, nous semble être une approche viable pour utiliser le *machine learning* dans les calculs, telle que la simulation physique, qui ne tolèrent pas les erreurs imprédictibles et le manque de contrôle sur les sorties, problématique pourtant assez courante dans les algorithmes d'apprentisage statistique.

Conclusion

Nous avons commencé ce doctorat avec l'objectif de concevoir des méthodes pour aider les développeurs à prendre des décisions averties sur la question du compromis entre précision et performance pour l'implémentation d'applications numériques. La possibilité de modifier la précision afin d'améliorer les performances fait l'objet d'une attention croissante maintenant que la loi de Moore atteint ses limites et qu'on ne peut pas s'attendre à obtenir des améliorations significatives des performances avec de nouveaux processeurs. Cela crée un besoin de méthodes pour étudier la précision et les performances dans les applications haute performance.

La majeure partie de cette thèse s'est concentrée sur la mesure de la précision numérique et, si une instabilité est détectée, sur la recherche des sources d'erreur numérique. C'est le travail que je vais passer en revue ici.

Je mets de côté notre travail de prédiction du profil de convergence d'un solveur linéaire car il est d'une nature différente. Toutefois, comme indiqué dans le chapitre 9.6, maintenant que nous avons montré qu'il est possible de mesurer des valeurs très simples et de produire un prédiction suffisamment précise pour améliorer à la fois la durée d'exécution et la précision d'un résultat en sélectionnant un solveur linéaire et un préconditionneur appropriés, je pense que ce domaine de recherche mérite d'être étudié plus avant.

Les développeurs qui veulent échanger de la précision contre de la performance disposent d'une vaste gamme d'outils pour mesurer la performance. Cependant, ils manquent d'outils pour mesurer la précision dans un environnement de calcul haute performance. Comme nous l'avons décrit dans le chapitre 3, l'arithmétique de haute précision, l'arithmétique des intervalles, l'arithmétique stochastique et l'analyse locale ont toutes été utilisées pour mesurer la précision numérique d'une application. Néanmoins, aucune de ces méthodes ne combine un surcoût suffisamment faible pour être appliquée à des calculs de longue durée, une précision sur un grand nombre d'opérations, une compatibilité avec le parallélisme et des résultats interprétables par un utilisateur qui est un expert du domaine (par exemple un physicien) plutôt qu'un spécialiste du calcul en virgule flottante. Toutes ces propriétés seraient souhaitables dans une méthode destinée à être utilisée par les développeurs d'applications haute performance et de simulations numériques.

Ma contribution à l'analyse du calcul en virgule flottante est la mise au point d'une nouvelle méthode, l'*erreur encapsulée*, conçue pour être utilisée dans le cadre du calcul haute performance et pour donner des résultats interprétables. Grâce à cette méthode, nous pouvons mesurer le nombre de chiffres significatifs de n'importe quel nombre et à n'importe quel point dans un calcul qui peut être en précision mixte ou évalué en parallèle.

Il semble important de souligner que notre travail n'est pas applicable à tous les cas d'utilisation, comme la vérification formelle de fonctions mathématiques (une application courante pour l'analyse du calcul à virgule flottante), mais qu'il est bien adapté à l'étude de larges simulations comme illustré dans le chapitre 8 de cette thèse.

Comme notre méthode nécessite des modifications du code source, nous avons rapidement rencontré des problèmes de passage à l'échelle lorsque nous avons voulu évaluer des applications trop larges pour être instrumentées manuellement. Notre solution a consisté à développer un outil de refactorisation du code, basé sur le compilateur Clang, capable de changer les types utilisés dans une application avec une intervention humaine minimale. Bien que nous soyons encore limités aux applications écrites dans des langages de programmation pour lesquels une implémentation de l'erreur encapsulée a été prévue, nous pouvons maintenant instrumenter des applications de milliers de lignes. Cela nous a permis de tester

des simulations physiques qui sont actuellement utilisées par les physiciens (comme nous l'avons vu dans le chapitre 8) en plus des problèmes jouets. Ce faisant, et en travaillant sur des problèmes de taille croissante, nous avons réalisé que nous ne pouvons pas nous attendre à ce que l'erreur numérique augmente de manière linéaire avec la taille du problème. À la place, nous observons une chute soudaine du nombre de chiffres significatifs (comme nous l'avons vu dans la section 8.3) alors que la perte de précision est lègère et progressive au début. Cela rend délicat le test de la précision d'une application, car les cas de test de petite ou moyenne taille peuvent ne pas déclencher cette chute, ce qui conduirait l'utilisateur à extrapoler et croire que son application est numériquement stable pour des cas de test plus larges.

Notre dernière contribution est un moyen de déterminer les sources des erreurs numériques observées. La nécessité d'un tel outil est apparue naturellement lorsque nous avons rencontré un programme numériquement instable mais trop large pour nous permettre de raisonner sur des opérations individuelles.

Notre solution a été le développement de l'*erreur taguée*, une variante de l'erreur encapsulée dans laquelle nous gardons les sources d'erreur définies par l'utilisateur séparées les unes des autres afin de pouvoir évaluer leur contribution individuelle au résultat final. Ainsi, un utilisateur peut taguer les composants principaux de son application et obtenir un rapport sur leur stabilité numérique individuelle et leur impact sur les calculs ultérieurs.

L'erreur taguée nous a appris, comme nous l'avons vu dans le chapitre 8, que l'erreur numérique ne vient pas toujours de là où nous l'attendons. L'intuition nous dit que la partie la plus complexe d'une application devrait être la source de l'erreur numérique, mais les algorithmes complexes (tels que la résolution de systèmes linéaires ou une décomposition en valeurs propres) ont tendance à s'appuyer sur des bibliothèques et des algorithmes qui ont été renforcés numériquement avec le temps. En revanche, des étapes d'initialisation conceptuellement simples ne bénéficient généralement pas de ces années de perfectionnement et, étonnamment, peuvent devenir des sources majeures d'erreurs numériques dans le résultat final. Une fois détectées, en raison de leur simplicité algorithmique, ces dernières sont un moyen simple d'améliorer la précision numérique d'une application en utilisant, par exemple, des algorithmes compensés.

Un autre aspect intéressant découlant de l'erreur taguée est la possibilité d'effectuer des améliorations numériques ciblées, en sacrifiant un minimum de performances pour une précision accrue. Un exemple direct est notre travail sur l'algorithme du gradient conjugué dans la section 6.3. Lorsque nous utilisons un solveur linéaire, nous pouvons employer le résidu comme mesure de précision mais, ayant la capacité de localiser les sources d'erreur, nous pouvons injecter des opérations compensées ciblées afin d'échanger de la performance contre une

précision supplémentaire sans avoir à repenser l'algorithme ou à introduire des éléments supplémentaires tels qu'un préconditionneur.

Tout notre travail est open-source et nous avons mis en ligne à la fois notre implémentation de référence, la bibliothèque Shaman, et les outils que nous avons développés [Demeure and Chevalier, 2019, Demeure and Ancellin, 2020]. Shaman est actuellement utilisé au CEA, à la fois pour valider des codes de simulation et pour évaluer les sorties d'applications qui ont été compilées avec un autre compilateur ou sur une nouvelle architecture de processeur. Il figure également dans des publications à venir sur les méthodes de quantification de l'incertitude et son utilisation est explorée par des chercheurs travaillant sur l'introduction de la précision mixte dans les calculs. Dans l'état actuel des choses, il serait intéressant d'investir des efforts d'ingénierie dans ces travaux pour produire des implémentations performantes couplées à des outils d'instrumentation pour les divers langages de programmation couramment utilisés en simulation, tels que Python, Matlab et Fortran. Une implémentation basée sur Valgrind, en particulier, faciliterait grandement l'instrumentation de codes utilisant plusieurs langages de programmation. Du point de vue de la recherche, deux directions se détachent. Premièrement, l'erreur taguée bénéficierait probablement de raffinements algorithmiques et de l'introduction d'une représentation interne efficace, à l'instar des travaux qui ont été réalisés sur l'arithmétique affine. Actuellement, si l'utilisateur a défini dix tags, alors tous les nombres auront dix termes d'erreur même s'il peut être démontré que la plupart de ces termes d'erreur sont nuls. Réduire les pertes dans la représentation pourrait conduire à une amélioration significative des performances. Deuxièmement, nous pensons pouvoir utiliser la possibilité d'accéder à l'erreur numérique *dans le code* comme une brique de construction pour construire des algorithmes numérique et des outils d'analyse nouveaux. Par exemple, nous pourrions utiliser notre connaissance de l'erreur numérique dans le critère d'arrêt d'un algorithme itératif.

Enfin, je voudrais souligner que je me suis concentré à dessein sur une solution simple. Bien que des solutions plus complexes puissent être séduisantes et malgré les pressions qui peuvent exister pour se concentrer sur des approches plus complexes, je crois fermement que les idées simples sont à la fois plus résistantes, ayant moins de parties succeptible d'échouer, et beaucoup plus susceptibles d'être réutilisées, étant faciles à mettre en œuvre et à reproduire. Idéalement, l'introduction d'une solution simple, précise, plus rapide que les méthodes existantes et facile à utiliser encouragera les auteurs de codes numériques à étudier la précision de leur calculs à virgule flottante et ne plus en négliger les effets.

université PARIS-SACLAY

ÉCOLE DOCTORALE
de mathématiques
Hadamard (EDMH)

Titre: Compromis entre précision et performance dans le calcul haute performance.

Mots clés: Arithmétique en virgule flottante, vérification numérique, erreur d'arrondi

Résumé: Les nombres à virgule flottante ne représentent qu'un sous-ensemble des nombres réels. De ce fait, l'arithmétique à virgule flottante introduit des approximations qui sont susceptibles de se cumuler et d'avoir un impact significatif sur les simulations numériques. Nous introduisons une nouvelle façon d'estimer et de localiser les sources d'erreur numérique dans une application et fournissons une implémentation de référence, la bibliothèque Shaman. Notre méthode utilise une arithmétique dédiée sur un type qui encapsule à la fois le résultat des calculs (identique à la version non instrumentée du code) et une approximation de son erreur numérique. Nous pouvons ainsi mesurer le nombre de chiffres significatifs de tout résultat ou résultat intermédiaire dans une simulation. Nous montrons que notre approche, bien que simple, donne des résultats compétitifs avec l'état de l'art. Qui plus est, elle a un surcoût en temps de calcul moins important et est compatible avec le parallélisme, ce qui la rend appropriée pour l'étude de larges applications.

Title: Compromise between precision and performance in high-performance computing.

Keywords: Floating-point arithmetic, numerical verification, round-off errors

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

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France