# Numerical validation and assessment of numerical accuracy*

Jean-Marie Chesneaux, Stef Graillat, Fabienne Jézéquel
Laboratoire d'Informatique de Paris 6, France

**Abstract**

Rounding errors present an inherent problem to all computer programs involving floating-point numbers. They appear nearly in all elementary operations. By accumulation, these errors can affect the accuracy of a computed result possibly leading to some or total inaccuracy. First in this article, computer arithmetic that uses binary system and the IEEE standard for floating-point numbers is described together with some problems introduced by rounding errors. The effects of these rounding errors can be analyzed and studied by some methods like forward/backward analysis, interval or stochastic arithmetic that are presented. Finally, it is shown how the accuracy of the results can be improved by using compensated methods and multiprecision arithmetic.

**Key words:** rounding error, floating-point arithmetic, IEEE 754 standard, accurate computation, interval arithmetic, stochastic arithmetic, multiprecision.

*This paper is an extended version of [9].

# Contents

# 1 Introduction

For a long time, human beings are in constant need of making bigger and faster computations. Over the past four centuries, many machines were created for this purpose and, fifty years ago, actual electronic computers were developed specifically to perform scientific computations. The first mechanical calculating machines were *Schikard's machine* (1623, Germany), the *Pascaline* (1642, France) followed by *Leibniz's machine* (1694, Germany). *Babbage's analytical machine* (1833, England) was the first attempt at a mechanical computer and the first mainframe computer was the *Z4 computer of K. Zuse* (1938, Germany).

Up until the beginning of the twentieth century, computations were only done on integer numbers. To perform efficient real numbers computations, it was necessary to wait until the birth of the famous BIT (*BInary digiT*) which was introduced by C. Shannon (1937, USA) in his PhD thesis. Shannon's work imposed electronics for the building of computers and, then, the base 2 for coding integer or real numbers, although other bases have been tried. It has now been established that the base 2 is the most efficient base on computers for numerical computations, although the base 10 may still be used on pocket calculators.

For coding real numbers, one also has to determine the kind of coding he wants to use. The decimal fixed-point notation was introduced at the end of the 16th century consecutively by S. Stévin (1582, France), J. Bürgi (1592, Switzerland) and G. Magini (1592, Italy). It remains the notation used worldwide today. While it is the most natural notation for mental calculations, it is not very efficient for automatic computations. In fact, on this subject, one can say that nothing has changed since J. Napier's logarithm (1614, Scotland) and W. Oughtred's slide rule (1622, England). Logarithms were introduced by J. Napier to make multiplication easier (using logarithm, multiplication becomes addition). Three centuries later, the same idea was kept for the coding of real numbers on computers and led to the floating-point representation (see next section).

But whatever the representation is on computer, it is a finite representation, like for computations by hand. So, at each operation, because the result needs to be truncated (but is in general), an error may appear which is called the rounding error. Scientists have been well aware of this for four centuries. In the 19th century, when numerical computations were presented in an article, they were systematically followed by errors computations to justify the validity of the results. In 1950, in his famous article on eigenvalue computation with his new algorithm, C. Lanczos devoted thirty percent of his paper to error computation. Unfortunately, this use has completely disappeared since the beginning of the sixties because of the improvement of computers. When 8 billion floating-point operations are performed in one second on a processor, it seems impossible to quantify the rounding error even though neglecting rounding errors may lead to catastrophic consequences.

For instance, for real time applications, the discretization step may be $h = 10^{-1}$ second. One can compute the absolute time by performing $t_{abs} = t_{abs} + h$ at each step or performing $icount = icount + 1; t_{abs} = h * icount$ where $icount$ is correctly initialized at the beginning of the process. Because the real number representation is finite on computers, only a finite number of them can be exactly coded. They are called floating-point numbers. The others are approximated by a floating-point number. Unfortunately, $h = 10^{-1}$ is not a floating-point number. Therefore, each operation $t_{abs} = t_{abs} + h$ generates a small but non-zero error. One hundred hours later, this error has grown to about 0.34 second. It really happened during the first Gulf war (1991) in the control programs of Patriot missiles which were to intercept Scud missiles (report of the General Accounting office, GAO/IMTEC-92-26). At 1600 km/h, 0.34 second corresponds to approximatively 500 meters, the interception failed and twenty-eight people were killed. With the second formulation, whatever the absolute time is, if no overflow occurs for *icount*, the relative rounding error remains below $10^{-15}$ using the IEEE double precision arithmetic. A good knowledge of the floating-point arithmetic should be required of all computer scientists [13].

The second section is devoted to the description of the computer arithmetic. The third section presents approaches to study, to bound or to estimate rounding errors. The last section describes methods to improve the accuracy of computed results. A goal of this paper is to answer the question in numerical computing *"What is the computing error due to floating-point arithmetic on the results produced by a program?"*

# 2 Computer arithmetic

## 2.1 Representation of numbers

In a numeral system, numbers are represented by a sequence of symbols. The number of distinct symbols which can be used is called the radix (or the base). For instance, in the decimal system, where the radix is 10, the ten symbols used are the digits 0,1,...,9. In the binary system, which is used on most computers, the radix is 2; hence numbers are represented with sequences of 0s and 1s.

Several formats exist to represent numbers on a computer. The representation of integer numbers differs from the one of real numbers. Using a radix $b$, if unsigned integers are encoded on $n$ digits, they can range from 0 to $b^n - 1$. Hence an unsigned integer $X$ is represented by a sequence $a_{n-1}a_{n-2} \dots a_1 a_0$ with $X = \sum_{i=0}^{n-1} a_i b^i$ and $a_i \in \{0, ..., b-1\}$. With a radix 2 representation, signed integers are usually represented using two's complement. With this rule, signed integers range from $-b^{n-1}$ to $b^{n-1} - 1$ and the sequence $a_{n-1}a_{n-2} \dots a_1 a_0$ with $a_i \in \{0, ..., b-1\}$ represents the number $X = -a_{n-1}b^{n-1} + \sum_{i=0}^{n-2} a_i b^i$. The opposite of a number in two's complement format can be obtained by inverting each bit and adding 1.

In numerical computations, most real numbers are not exactly represented because only a finite number of digits can be saved in memory. Two representations exist for real numbers:

- the fixed-point format, available on most embedded systems
- the floating-point format, available on classical computers.

In fixed-point arithmetic, a number is represented with a fixed number of digits before and after the radix point. Using a radix $b$, a number $X$ which is encoded on $m$ digits for its magnitude (*e.g.* its integer part) and $f$ digits for its fractional part is represented by $a_{m-1} \dots a_0 \,.\, a_{-1} \dots a_{-f}$, with $X = \sum_{i=-f}^{m-1} a_i b^i$ and $a_i \in \{0, ..., b-1\}$.

If $b = 2$, unsigned values range from 0 to $2^m - 2^{-f}$ and signed values, which are usually represented with the two's complement format, range from $-2^{m-1}$ to $2^{m-1} - 2^{-f}$.

In a floating-point arithmetic using the radix $b$, a number $X$ is represented by:

- its sign $\varepsilon_X$ which is encoded on one digit that equals 0 if $\varepsilon_X = 1$ and 1 if $\varepsilon_X = -1$,
- its exponent $E_X$, a $k$ digit integer,
- its mantissa $M_X$, encoded on $p$ digits.

Therefore $X = \varepsilon_X M_X b^{E_X}$ with $M_X = \sum_{i=0}^{p-1} a_i b^{-i}$ and $a_i \in \{0, ..., b-1\}$. The mantissa $M_X$ can be written as $M_X = a_0 \,.\, a_1 \,...\, a_{p-1}$. Floating-point numbers are usually normalized. In this case, $a_0 \neq 0$, $M_X \in [1, b)$ and the number zero has a special representation. Normalization presents several advantages, such as the uniqueness of the representation (there is exactly one way to write a number in such a form) and the easiness of comparisons (the signs, exponents and mantissas of two normalized numbers can be tested separately).

## 2.2 The IEEE 754 standard

The poor definition of the floating-point arithmetic on most computers created the need for a unified standard in floating-point numbers. Indeed the bad quality of arithmetic operators could heavily affect some results. Furthermore simulation programs could provide different results from one computer to another, because of different floating-point representations. Indeed different values could be used for the radix, the length of the exponent, the length of the mantissa, etc. So, in 1985, the IEEE 754 standard [18] was elaborated to define floating-point formats and rounding modes. It specifies two basic formats, both using the radix 2.

- With the single precision format, numbers are stored on 32 bits: 1 for the sign, 8 for the exponent and 23 for the mantissa.
- With the double precision format, numbers are stored on 64 bits: 1 for the sign, 11 for the exponent and 52 for the mantissa.

Extended floating-point formats also exist; the standard does not specify their exact size, but gives a minimum number of bits for their storage.

Because of the normalization, the first bit in the mantissa must be 1. As this implicit bit is not stored, the precision of the mantissa is actually 24 bits in single precision and 53 bits in double precision.

The exponent $E$ is a $k$ digit signed integer. Let us denote its bounds by $E_{min}$ and $E_{max}$. The exponent which is actually stored is a biased exponent $E_\Delta$ such that $E_\Delta = E + \Delta$, $\Delta$ being the bias. Table 1 specifies how the exponent is encoded.

The number zero is encoded by setting to 0 all the bits of the (biased) exponent and all the bits of the mantissa. Two representations actually exist for zero: $+0$ if the sign bit is 0 and $-0$ if the sign bit is 1. This distinction is consistent with the existence of two infinities.

| precision | length | bias | non-biased | | biased | |
|---|---|---|---|---|---|---|
| | $k$ | $\Delta$ | $E_{min}$ | $E_{max}$ | $E_{min}+\Delta$ | $E_{max}+\Delta$ |
| single | 8 | 127 | -126 | 127 | 1 | 254 |
| double | 11 | 1023 | -1022 | 1023 | 1 | 2046 |

Table 1: Exponent coding in single and double precision

Indeed $1/(+0) = +\infty$ and $1/(-0) = -\infty$. These two infinities are encoded by setting to 1 all the bits of the (biased) exponent and to 0 all the bits from the mantissa. The corresponding non-biased exponent is therefore $E_{max} + 1$.

NaN (Not a Number) is a special value which represents the result of an invalid operation such as $0/0$, $\sqrt{-1}$ or $0 \times \infty$. NaN is encoded by setting all the bits of the (biased) exponent to 1 and the fractional part of the mantissa to any non-zero value.

Denormalized numbers (also called subnormal numbers) represent values close to zero. Without them, as the integer part of the mantissa is implicitly set to 1, there would be no representable number between 0 and $2^{E_{min}}$ but $2^{p-1}$ representable numbers between $2^{E_{min}}$ and $2^{E_{min}+1}$. Denormalized numbers have a biased exponent set to 0. The corresponding values are: $X = \varepsilon_X M_X 2^{E_{min}}$ with $\varepsilon_X = \pm 1$, $M_X = \sum_{i=1}^{p-1} a_i 2^{-i}$ and $a_i \in \{0, 1\}$. The mantissa $M_X$ can be written as $M_X = 0.a_1...a_{p-1}$. Therefore the lowest positive denormalized number is $\underline{u} = 2^{E_{min}+1-p}$. Moreover, denormalized numbers and gradual underflow imply the nice equivalence $a = b \iff a - b = 0$.

Let us denote by $\mathbb{F}$ the set of all floating-point numbers, *i.e.* the set of all machine representable numbers. This set, which depends on the chosen precision, is bounded and discrete. Let us denote its bounds by $X_{min}$ and $X_{max}$. Let $x$ be a real number which is not machine representable. If $x \in (X_{min}, X_{max})$, there exists $\{X^-, X^+\} \subset \mathbb{F}^2$ such that $X^- < x < X^+$ and $(X^-, X^+) \cap \mathbb{F} = \emptyset$. A rounding mode is a rule which, from $x$, provides $X^-$ or $X^+$. This rounding occurs at each assignment and at each arithmetic operation. The IEEE 754 standard imposes a correct rounding for all arithmetic operations ($+$, $-$, $\times$, $/$) and also for the square root. The result must be the same as the one obtained with infinite precision and then rounded. The IEEE 754 standard defines four rounding modes:

- rounding towards $+\infty$ (or upward rounding), $x$ is represented by $X^+$,
- rounding towards $-\infty$ (or downward rounding), $x$ is represented by $X^-$,
- rounding towards 0, if $x$ is negative, then it is represented by $X^+$, if $x$ is positive, then it is represented by $X^-$,
- rounding to nearest, $x$ is represented by its nearest machine number. If $x$ is at the same distance of $X^-$ and $X^+$, it is represented by the machine number which has a mantissa ending with a zero. With this rule, rounding is said to be *tie to even*.

Let us denote by $X$ the number obtained by applying one of these rounding modes to $x$. By definition, an *overflow* occurs if $|X| > \max\{|Y| : Y \in \mathbb{F}\}$ and an *underflow* occurs if

$0 < |X| < \min\{|Y| : 0 \neq Y \in \mathbb{F}\}$. Gradual underflow denotes the situation where a number is not representable as a normalized number, but still as a denormalized one.

## 2.3 Rounding error formalization

### 2.3.1 Notion of exact significant digits

In order to correctly quantify the accuracy of a computed result, the notion of exact significant digits must be formalized. Let $R$ be a computed result and $r$ the corresponding exact result. The number $C_{R,r}$ of exact significant decimal digits of $R$ is defined as the number of significant digits which are in common with $r$:

$$C_{R,r} = \log_{10} \left| \frac{R+r}{2(R-r)} \right|. \tag{2.1}$$

This mathematical definition is in accordance with the intuitive idea of decimal significant digits in common between two numbers. Indeed Equation (2.1) is equivalent to

$$|R-r| = \left| \frac{R+r}{2} \right| 10^{-C_{R,r}}. \tag{2.2}$$

If $C_{R,r} = 3$, the relative error between $R$ and $r$ is of the order of $10^{-3}$. $R$ and $r$ have therefore three common decimal digits.

However the value of $C_{R,r}$ may seem surprising if one considers the decimal notations of $R$ and $r$. For example, if $R = 2.4599976$ and $r = 2.4600012$, then $C_{R,r} \approx 5.8$. The difference due to the sequences of “0” or “9” is illusive. The significant decimal digits of $R$ and $r$ are really different from the sixth position.

### 2.3.2 Rounding error occurring at each operation

A formalization of rounding errors generated by assignments and arithmetic operations is proposed below. Let $X$ be the representation of a real number $x$ in a floating-point arithmetic respecting the IEEE 754 standard. This floating-point representation of $X$ may be written as $X = \mathrm{fl}(x)$. Adopting the same notations as in 2.1,

$$X = \varepsilon_X M_X 2^{E_X} \tag{2.3}$$

and

$$X = x - \varepsilon_X 2^{E_X - p} \alpha_X \tag{2.4}$$

where $\alpha_X$ represents the normalized rounding error.

- with rounding to nearest, $\alpha_X \in [-0.5, 0.5)$
- with rounding towards zero, $\alpha_X \in [0, 1)$
- with rounding towards $+\infty$ or $-\infty$, $\alpha_X \in [-1, 1)$.

Equivalent models for $X$ are given below. The *machine epsilon* is the distance $\epsilon$ from 1.0 to the next larger floating-point number. Clearly, $\epsilon = 2^{1-p}$, $p$ being the length of the mantissa including the implicit bit. The relative error on $X$ is no larger than the *unit round-off* $u$:

$$X = x(1+\delta) \text{ with } |\delta| \leq u \tag{2.5}$$

where $u = \epsilon/2$ with rounding to nearest and $u = \epsilon$ with the other rounding modes. The model associated with Equation (2.5) ignores the possibility of underflow. To take underflow into account, one must modify it to

$$X = x(1+\delta) + \eta \text{ with } |\delta| \leq u \tag{2.6}$$

and $|\eta| \leq \underline{u}/2$ with rounding to nearest and $|\eta| \leq \underline{u}$ with the other rounding modes, $\underline{u}$ being the lowest positive denormalized number.

Let $X_1$ (respectively $X_2$) be the floating-point representation of a real number $x_1$ (respectively $x_2$)

$$X_i = x_i - \varepsilon_i 2^{E_i - p}\alpha_i \quad \text{for} \quad i = 1, 2. \tag{2.7}$$

The errors due to arithmetic operations having $X_1$ and $X_2$ as operands are given below. For each operation, let us denote by $E_3$ and $\varepsilon_3$ the exponent and the sign of the computed result. $\alpha_3$ represents the rounding error performed on the result. Let us denote by $\oplus$, $\ominus$, $\otimes$, $\oslash$ the arithmetic operators on a computer.

$$X_1 \oplus X_2 = x_1 + x_2 - \varepsilon_1 2^{E_1-p}\alpha_1 - \varepsilon_2 2^{E_2-p}\alpha_2 - \varepsilon_3 2^{E_3-p}\alpha_3. \tag{2.8}$$

Similarly

$$X_1 \ominus X_2 = x_1 - x_2 - \varepsilon_1 2^{E_1-p}\alpha_1 + \varepsilon_2 2^{E_2-p}\alpha_2 - \varepsilon_3 2^{E_3-p}\alpha_3. \tag{2.9}$$

$$X_1 \otimes X_2 = x_1 x_2 - \varepsilon_1 2^{E_1-p}\alpha_1 x_2 - \varepsilon_2 2^{E_2-p}\alpha_2 x_1 + \varepsilon_1\varepsilon_2 2^{E_1+E_2-2p}\alpha_1\alpha_2 - \varepsilon_3 2^{E_3-p}\alpha_3. \tag{2.10}$$

By neglecting the fourth term, which is of the second order in $2^{-p}$, one obtains

$$X_1 \otimes X_2 = x_1 x_2 - \varepsilon_1 2^{E_1-p}\alpha_1 x_2 - \varepsilon_2 2^{E_2-p}\alpha_2 x_1 - \varepsilon_3 2^{E_3-p}\alpha_3. \tag{2.11}$$

By neglecting terms of an order greater than or equal to $2^{-2p}$, one obtains

$$X_1 \oslash X_2 = \frac{x_1}{x_2} - \varepsilon_1 2^{E_1-p}\frac{\alpha_1}{x_2} + \varepsilon_2 2^{E_2-p}\alpha_2\frac{x_1}{x_2^2} - \varepsilon_3 2^{E_3-p}\alpha_3. \tag{2.12}$$

In the case of an addition with operands of the same sign,

$$E_3 = \max(E_1, E_2) + \delta \text{ with } \delta = 0 \text{ or } \delta = 1.$$

The order of magnitude of the two terms resulting from the rounding errors on $X_1$ and $X_2$ is at most $2^{E_3-p}$. The relative error on $X_1 \oplus X_2$ remains of the order of $2^{-p}$. This operation is therefore relatively stable: it does not induce any brutal loss of accuracy.

The same conclusions are valid in the case of a multiplication, because

$$E_3 = E_1 + E_2 + \delta, \text{ with } \delta = 0 \text{ or } \delta = -1$$

and in the case of a division, because

$$E_3 = E_1 - E_2 + \delta, \text{ with } \delta = 0 \text{ or } \delta = 1.$$

In the case of a subtraction with operands of the same sign, $E_3 = \max(E_1, E_2) - k$. If $X_1$ and $X_2$ are very close, $k$ may be large. The order of magnitude of the absolute error remains $2^{\max(E_1,E_2)-p}$, but the order of magnitude of the relative error is $2^{\max(E_1,E_2)-p-E_3} = 2^{-p+k}$. In one operation, $k$ exact significant bits have been lost: it is the so-called *catastrophic cancellation.*

### 2.3.3 Rounding error propagation

A numerical program is a sequence of arithmetic operations. The result $R$ provided by a program after $n$ operations or assignments can be modelled to the first order in $2^{-p}$ as:

$$R \approx r + \sum_{i=1}^{n} g_i(d) 2^{-p} \alpha_i \tag{2.13}$$

where $r$ is the exact result, $p$ is the number of bits in the mantissa, $\alpha_i$ are independent uniformly distributed random variables on $[-1, 1]$ and $g_i(d)$ are coefficients depending exclusively on the data and on the code [7]. For instance, in Equation (2.12), $g_i(d)$ are $\frac{1}{x_2}$ and $\frac{x_1}{x_2^2}$.

The number $C_{R,r}$ of exact significant bits of the computed result $R$ is

$$C_{R,r} = \log_2 \left| \frac{R + r}{2(R - r)} \right|. \tag{2.14}$$

$$C_{R,r} \approx -\log_2 \left| \frac{R - r}{r} \right| = p - \log_2 \left| \sum_{i=1}^{n} g_i(d) \frac{\alpha_i}{r} \right|. \tag{2.15}$$

The last term in Equation (2.15) represents the loss of accuracy in the computation of $R$. This term is independent of $p$. Therefore, assuming that the model at the first order established in Equation (2.13) is valid, the loss of accuracy in a computation is independent of the precision used.

## 2.4 Impact of rounding errors on numerical programs

With floating-point arithmetic, rounding errors occur in numerical programs and lead to a loss of accuracy which is difficult to estimate. Another consequence of floating-point arithmetic is the loss of algebraic properties. The floating-point addition and the floating-point multiplication are commutative, but not associative. Therefore the same formula may generate different results depending on the order in which arithmetic operations are executed. For instance, in IEEE single precision arithmetic with rounding to nearest,

$$(-10^{20} \oplus 10^{20}) \oplus 1 = 1 \tag{2.16}$$

but

$$-10^{20} \oplus (10^{20} \oplus 1) = 0. \tag{2.17}$$

Equation (2.17) causes a so-called *absorption*. Indeed an absorption may occur during the addition of numbers with very different orders of magnitude: the smallest number may be lost.

Furthermore, with floating-point arithmetic, the multiplication is not distributive with respect to the addition. Let $A$, $B$ and $C$ be floating-point numbers, $A \otimes (B \oplus C)$ may not be equal to $(A \otimes B) \oplus (A \otimes C)$. For instance, in IEEE single precision arithmetic with rounding to nearest, if $A$, $B$ and $C$ are respectively assigned to 3.3333333, 12345679 and 1.2345678, for $A \otimes (B \oplus C)$ and $(A \otimes B) \oplus (A \otimes C)$, one respectively obtains 41152264 and 41152268.

### 2.4.1 Impact on direct methods

The particularity of a direct method is to provide the solution to a problem in a finite number of steps. In infinite precision arithmetic, a direct method would compute the exact result. In finite precision arithmetic, rounding error propagation induces a loss of accuracy and may cause problems in branching statements. The general form of a branching statement in a program is

IF condition THEN sequence 1 ELSE sequence 2.

If the condition is satisfied, then a sequence of instructions is executed, else another sequence is performed. Such a condition can be for instance $A \geq B$. In the case when $A$ and $B$ are intermediate results already affected by rounding errors, the difference between $A$ and $B$ may have no exact significant digit. The choice of the sequence which is executed may depend on rounding error propagation. The sequence chosen may be the wrong one: it may be different from the one that would have been chosen in exact arithmetic.

For instance, depending on the value of the discriminant, a second degree polynomial has one (double) real root, two real roots or two conjugate complex roots. The discriminant and the roots of the polynomial $0.3x^2 - 2.1x + 3.675$ obtained using IEEE single precision arithmetic with rounding to nearest are $D = -5.185604\text{E-}07$, $x = 3.4999998 \pm 1.2001855\text{E-}03\ i$. Two conjugate complex roots are computed. But the exact values are $D = 0$, $x = 3.5$. The polynomial actually has one double real root. In floating-point arithmetic, rounding errors occur because of both assignments and arithmetic operations. Indeed the coefficients of the polynomial are not floating-point numbers. Therefore the computed discriminant has no exact significant digit and the wrong sequence of instructions is executed.

### 2.4.2 Impact on iterative methods

The result of an iterative method is defined as the limit $L$ of a first order recurrent sequence:

$$L = \lim_{n \to \infty} U_n \quad \text{with} \quad U_{n+1} = \mathcal{F}(U_n) \quad \mathbb{R}^m \xrightarrow{\mathcal{F}} \mathbb{R}^m. \tag{2.18}$$

Because of rounding error propagation, the same problems as in a direct method may occur. But another difficulty is due to the loss of the notion of limit on a computer. Computations are performed until a stopping criterion is satisfied. Such a stopping criterion may involve the absolute error:

$$\|U_n - U_{n-1}\| \leq \varepsilon \tag{2.19}$$

or the relative error:

$$\|U_n - U_{n-1}\| \leq \varepsilon \|U_{n-1}\| . \tag{2.20}$$

It may be difficult to choose a suitable value for $\varepsilon$. If $\varepsilon$ is too high, computations stop too early and the result is very approximative. If $\varepsilon$ is too low, useless iterations are performed without improving the accuracy of the result, because of rounding errors. In this case, the stopping criterion may never be satisfied because the chosen accuracy is illusive. The impact of $\varepsilon$ on the quality of the result is shown in the numerical experiment described below.

Newton's method is used to compute a root of

$$f(x) = x^4 - 1002x^3 + 252001x^2 - 501000x + 250000. \tag{2.21}$$

The following sequence is computed:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad \text{with} \quad x_0 = 1100. \tag{2.22}$$

The exact limit is $L = 500$, a double root of $f$. The stopping criterion is $|x_n - x_{n-1}| \leq \varepsilon |x_{n-1}|$ and the maximum number of iterations is set to 1000. Table 2 shows for several values of $\varepsilon$ the last value of $n$ and the error $|x_n - L|$ computed using IEEE double precision arithmetic with rounding to nearest.

| $\varepsilon$ | $n$ | $\|x_n - L\|$ |
|---|---|---|
| $10^{-7}$ | 26 | 3.368976E-05 |
| $10^{-8}$ | 29 | 4.211986E-06 |
| $10^{-9}$ | 33 | 2.525668E-07 |
| $10^{-10}$ | 35 | 1.405326E-07 |
| $10^{-11}$ | 127 | 1.273870E-07 |
| $10^{-12}$ | 1000 | 1.573727E-07 |
| $10^{-13}$ | 1000 | 1.573727E-07 |

Table 2: Number of iterations and error obtained using Newton's method in double precision

It appears that the optimal order of magnitude for $\varepsilon$ is $10^{-11}$. The stopping criterion can not be satisfied if $\varepsilon \leq 10^{-12}$: the maximum number of iterations is reached; furthermore the error is slightly higher than for $\varepsilon = 10^{-11}$.

The difficulty to choose a suitable stopping criterion can also be exemplified by the dichotomy method applied to a polynomial $P$. Indeed the stopping criterion $|P(x)| \leq \varepsilon$ may lead to an infinite loop for unappropriate values of $\varepsilon$. For instance, the value obtained in IEEE single precision arithmetic using rounding to nearest for

$$P(x) = x^3 - 111x^2 + 1100x - 3000 \tag{2.23}$$

with $x_1 = 100.334694$ is $P(x_1) = -0.014937866$. The value obtained for $x_2$, the floating-point successor of $x_1$ with this arithmetic is $P(x_2) = 0.05393155$. Therefore the stopping criterion cannot be satisfied if $\varepsilon \leq 10^{-2}$.

### 2.4.3 Impact on approximation methods

These methods provide an approximation of a limit $L = \lim_{h\to 0} L(h)$. This approximation is affected by a global error $E_g(h)$ which consists in a truncation error $E_t(h)$, inherent to the method, and a rounding error $E_r(h)$. If the step $h$ decreases, the truncation error $E_t(h)$ also decreases, but the rounding error $E_r(h)$ usually increases, as shown in Figure 1. It may therefore seem difficult to choose the optimal step $h_{opt}$. The rounding error should be evaluated, because the global error is minimal if the truncation error and the rounding error have the same order of magnitude.

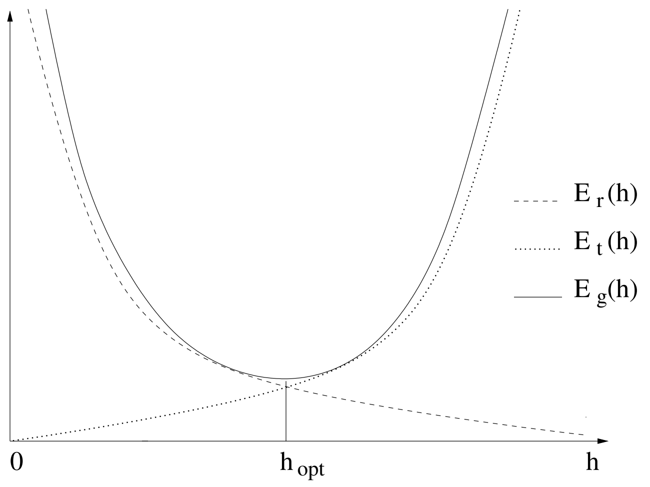

Figure 1: Evolution of the rounding error $E_r(h)$, the truncation error $E_t(h)$ and the global error $E_g(h)$ with respect to the step $h$

The numerical experiment described below [32] shows the impact of the step $h$ on the quality of the approximation. The second derivative at $x = 1$ of the following function

$$f(x) = \frac{4970x - 4923}{4970x^2 - 9799x + 4830} \tag{2.24}$$

is approximated by

$$L(h) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}. \tag{2.25}$$

The exact result is $f''(1) = 94$. Table 3 shows for several steps $h$ the result $L(h)$ and the absolute error $|L(h) - L|$ computed using IEEE double precision arithmetic with rounding to nearest.

| $h$ | $L(h)$ | $\lvert L(h) - L\rvert$ |
|---|---|---|
| $10^{-3}$ | -2.250198E+03 | 2.344198E+03 |
| $10^{-4}$ | 7.078819E+01 | 2.321181E+01 |
| $10^{-5}$ | 9.376629E+01 | 2.337145E-01 |
| $4.10^{-6}$ | 9.397453E+01 | 2.546980E-02 |
| $3.10^{-6}$ | 9.397742E+01 | 2.257732E-02 |
| $10^{-6}$ | 9.418052E+01 | 1.805210E-01 |
| $10^{-7}$ | 7.607526E+01 | 1.792474E+01 |
| $10^{-8}$ | 1.720360E+03 | 1.626360E+03 |
| $10^{-9}$ | -1.700411E+05 | 1.701351E+05 |
| $10^{-10}$ | 4.111295E+05 | 4.110355E+05 |

Table 3: Second order approximation of $f''(1) = 94$ computed in double precision

It appears that the optimal order of magnitude for $h$ is $10^{-6}$. If $h$ is too low, the rounding error prevails and invalidates the computed result.

The computation of the derivative at $x = 100$ of the polynomial $P$ given in Equation (2.23) also exemplifies the loss of limits in floating-point arithmetic. $P'(100)$, the exact value of which is 8900, is defined as

$$P'(100) = \lim_{h \to 0} \frac{P(100+h) - P(100)}{h}. \tag{2.26}$$

If $P'(100)$ is computed using Equation (2.26), a null value may be obtained. Indeed, using IEEE single precision arithmetic with rounding to nearest, if $h \leq 10^{-6}$, $P(100+h) = P(100)$. This equality is also satisfied in double precision if $h \leq 10^{-8}$.

## 3 Methods for rounding error analysis

In this section, different methods of analyzing rounding errors are reviewed.

### 3.1 Forward/Backward analysis

This subsection is heavily inspired from [16, 23]. Other good references are [6, 33, 39].

Let $X$ be an approximation to a real number $x$. The two common measures of the accuracy of $X$ are its *absolute error*

$$E_a(X) = |x - X|, \tag{3.27}$$

and its *relative error*

$$E_r(X) = \frac{|x - X|}{|x|} \tag{3.28}$$

(which is undefined if $x = 0$). When $x$ and $X$ are vectors, the relative error is usually defined with a norm as $\|x - X\|/\|x\|$. This is a *normwise relative error.* A more widely used relative error is the *componentwise relative error* defined by $\max_i \frac{|x_i - X_i|}{|x_i|}$. It makes it possible to put the individual relative errors on an equal footing.

### 3.1.1 Well-posed problems

Let us consider the following mathematical problem $(P)$

$$(P) : \text{given } y, \text{ find } x \text{ such that } F(x) = y,$$

where $F$ is a continuous mapping between two linear spaces (in general $\mathbb{R}^n$ or $\mathbb{C}^n$). One will say that the problem $(P)$ is *well-posed* in the sense of Hadamard if the solution $x = F^{-1}(y)$ exists, is unique and $F^{-1}$ is continuous in the neighborhood of $y$. If it is not the case, one says that the problem is *ill-posed*. An example of ill-posed problem is the solution of a linear system $Ax = b$ where $A$ is singular. It is difficult to deal numerically with ill-posed problems (this is generally done via regularization techniques). That is why we will focus only on well-posed problems in the sequel.

### 3.1.2 Conditioning

Given a well-posed problem $(P)$, one wants now to know how to measure the difficulty of solving this problem. This will be done via the notion of *condition number*. Roughly speaking, the condition number measures the sensitivity of the solution to perturbation in the data. Since the problem $(P)$ is well-posed, one can write it as $x = G(y)$ with $G = F^{-1}$.

The input space (data) and the output space (result) are denoted respectively by $\mathcal{D}$ and $\mathcal{R}$; the norms on these spaces will be denoted $\|\cdot\|_{\mathcal{D}}$ and $\|\cdot\|_{\mathcal{R}}$. Given $\varepsilon > 0$ and let $\mathcal{P}(\varepsilon) \subset \mathcal{D}$ be a set of perturbation $\Delta y$ of the data $y$ satisfying $\|\Delta y\|_{\mathcal{D}} \le \varepsilon$, the perturbed problem associated with problem $(P)$ is defined by

$$\text{Find } \Delta x \in \mathcal{R} \text{ such that } F(x + \Delta x) = y + \Delta y \text{ for a given } \Delta y \in \mathcal{P}(\varepsilon).$$

$x$ and $y$ are assumed to be non-zero. The *condition number* of the problem $(P)$ in the data $y$ is defined by

$$\operatorname{cond}(P, y) := \lim_{\varepsilon \to 0} \sup_{\Delta y \in \mathcal{P}(\varepsilon), \Delta y \neq 0} \left\{ \frac{\|\Delta x\|_{\mathcal{R}}}{\|\Delta y\|_{\mathcal{D}}} \right\}. \tag{3.29}$$

**Example 3.1** (summation)**.** Let us consider the problem of computing the sum $x = \sum_{i=1}^{n} y_i$ assuming that $y_i \neq 0$ for all $i$. One will take into account the perturbation of the input data that are the coefficients $y_i$. Let $\Delta y = (\Delta y_1, \ldots, \Delta y_n)$ be the perturbation on $y = (y_1, \ldots, y_n)$. It follows that $\Delta x = \sum_{i=1}^{n} \Delta y_i$. Let us endow $\mathcal{D} = \mathbb{R}^n$ with the relative norm $\|\Delta y\|_{\mathcal{D}} = \max_{i=1,\ldots,n} |\Delta y_i|/|y_i|$ and $\mathcal{R} = \mathbb{R}$ with the relative norm $\|\Delta x\|_{\mathcal{R}} = |\Delta x|/|x|$. Since

$|\Delta x| = |\sum_{i=1}^n \Delta y_i| \leq \|\Delta y\|_{\mathcal{D}} \sum_{i=1}^n |y_i|$[1], one has

$$\frac{\|\Delta x\|_{\mathcal{R}}}{\|\Delta y\|_{\mathcal{D}}} \leq \frac{\sum_{i=1}^n |y_i|}{|\sum_{i=1}^n y_i|}. \tag{3.30}$$

This bound is reached for the perturbation $\Delta y$ such that $\Delta y_i / y_i = \text{sign}(y_i)\|\Delta y\|_{\mathcal{D}}$ where sign is the sign of a real number. As a consequence,

$$\text{cond}\left(\sum_{i=1}^n y_i\right) = \frac{\sum_{i=1}^n |y_i|}{|\sum_{i=1}^n y_i|}. \tag{3.31}$$

Now one has to interpret this condition number. A problem is considered as ill-conditioned if it has a large condition number. Otherwise, it is well-conditioned. It is difficult to give a precise frontier between well-conditioned and ill-conditioned problems. This statement will be clarified in 3.1.4 thanks to the rule of thumb. The larger the condition number is, the more a small perturbation on the data can imply a greater error on the result. Nevertheless, the condition number measures the *worst case* implied by a small perturbation. As a consequence, it is possible for an ill-conditioned problem that a small perturbation on the data also implies a small perturbation on the result. Sometimes, such a behavior is even typical.

*Remark* 1. It is important to note that the condition number is independent of the algorithm used to solve the problem. It is only a characteristic of the problem.

### 3.1.3 Stability of an algorithm

Problems are generally solved using an *algorithm*. This is a set of operations and tests that one can consider as the function $G$ defined above given the solution of our problem. Because of the rounding errors, the algorithm is not the function $G$ but rather a function $\widehat{G}$. Therefore, the algorithm does not compute $x = G(y)$ but $\widehat{x} = \widehat{G}(y)$.

The *forward analysis* tries to study the execution of the algorithm $\widehat{G}$ on the data $y$. Following the propagation of the rounding errors in each intermediate variables, the forward analysis tries to estimate or to bound the difference between $x$ and $\widehat{x}$. This difference between the exact solution $x$ and the computed solution $\widehat{x}$ is called the *forward error*.

It is easy to see that it is pretty difficult to follow the propagation of all the intermediate rounding errors. The *backward analysis* makes it possible to avoid this problem by working with the function $G$ itself. The idea is to seek for a problem that is actually solved and to check if this problem is "close to" the initial one. Basically, one tries to put the error on the result as an error on the data. More theoretically, one seeks for $\Delta y$ such that $\widehat{x} = G(y + \Delta y)$. $\Delta y$ is said to be the *backward error* associated with $\widehat{x}$. A backward error measures the distance between the problem that is solved and the initial problem. As $\widehat{x}$ and $G$ are known, it is often possible to obtain a good upper bound for $\Delta y$ (generally, it is easier than for the forward error). Figure 2 sums up the principle of the forward and backward analysis.

[1]the Cauchy-Schwarz inequality $|\sum_{i=1}^n x_i y_i| \leq \max_{i=1,\dots,n} |x_i| \times \sum_{i=1}^n |y_i|$ is used

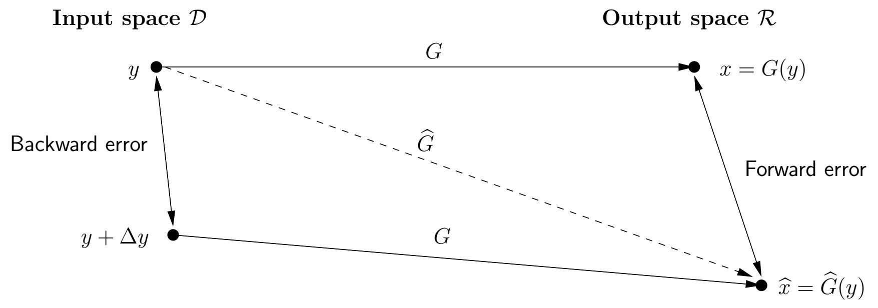
Input space $\mathcal{D}$
Output space $\mathcal{R}$
$y$
$G$
$x = G(y)$
Backward error
$\widehat{G}$
Forward error
$y + \Delta y$
$G$
$\widehat{x} = \widehat{G}(y)$

Figure 2: Forward and backward error for the computation of $x = G(y)$.

Sometimes, it is not possible to have $\widehat{x} = G(y + \Delta y)$ for some $\Delta y$ but it is often possible to get $\Delta x$ and $\Delta y$ such that $\widehat{x} + \Delta x = G(y + \Delta y)$. Such a relation is called a *mixed forward-backward error.*

The *stability* of an algorithm describes the influence of the computation in finite precision on the quality of the result. The backward error associated with $\widehat{x} = \widehat{G}(y)$ is the scalar $\eta(\widehat{x})$ defined by, when it exists,

$$\eta(\widehat{x}) = \min_{\Delta y \in \mathcal{D}} \{ \|\Delta y\|_{\mathcal{D}} : \widehat{x} = G(y + \Delta y) \}. \tag{3.32}$$

If it does not exist, $\eta(\widehat{x})$ is set to $+\infty$. An algorithm is said to be *backward-stable* for the problem $(P)$ if the computed solution $\widehat{x}$ has a "small" backward error $\eta(\widehat{x})$. In general, in finite precision, "small" means of the order of the rounding unit $u$.

**Example 3.2** (summation)**.** The addition is supposed to satisfy the following property:

$$\widehat{z} = z(1 + \delta) = (a + b)(1 + \delta), \quad \text{with } |\delta| \le u. \tag{3.33}$$

It should be noticed that this assumption is satisfied by the IEEE arithmetic. The following algorithm to compute the sum $\sum y_i$ will be used.

**Algorithm 3.1.** Computation of the sum of floating-point numbers

```
function res = Sum($y$)
  $s_1 = y_1$
  for $i = 2 : n$
    $s_i = s_{i-1} \oplus y_i$
  res = $s_n$
```

Thanks to relation (3.33), one can write

$$s_i = (s_{i-1} + y_i)(1 + \delta_i) \quad \text{with } |\delta_i| \le u. \tag{3.34}$$

For convenience, $1 + \theta_j = \prod_{i=1}^{j} (1 + \varepsilon_i)$ is written, for $|\varepsilon_i| \le u$ and $j \in \mathbb{N}$. Iterating the previous equation yields

$$\texttt{res} = y_1(1 + \theta_{n-1}) + y_2(1 + \theta_{n-1}) + y_3(1 + \theta_{n-2}) + \cdots + y_{n-1}(1 + \theta_2) + y_n(1 + \theta_1). \tag{3.35}$$

One can interpret the computed sum as the exact sum of the vector $z$ with $z_i = y_i(1+\theta_{n+1-i})$ for $i = 2 : n$ and $z_1 = y_1(1 + \theta_{n-1})$.

As $|\varepsilon_i| \leq u$ for all $i$ and assuming $nu < 1$, it can be proved that $|\theta_i| \leq iu/(1 - iu)$ for all $i$. Consequently, one can conclude that the backward error satisfies

$$\eta(\widehat{x}) = |\theta_{n-1}| \lesssim nu. \tag{3.36}$$

Since the backward error is of the order of $u$, one concludes that the classic summation algorithm is backward-stable.

### 3.1.4 Accuracy of the solution

How is the accuracy of the computed solution estimated? The accuracy of the computed solution actually depends on the condition number of the problem and on the stability of the algorithm used. The condition number measures the effect of the perturbation of the data on the result. The backward error simulates the errors introduced by the algorithm as errors on the data. As a consequence, at the first order, one has the following *rule of thumb*:

$$\boxed{\text{forward error } \lesssim \text{ condition number } \times \text{ backward error.}} \tag{3.37}$$

If the algorithm is backward-stable (that is to say the backward error is of the order of the rounding unit $u$) then the rule of thumb can be written as follows

$$\text{forward error } \lesssim \text{ condition number } \times \ u. \tag{3.38}$$

In general, the condition number is hard to compute (as hard as the problem itself). As a consequence, there are some estimators that make it possible to compute an approximation of the condition number with a reasonable complexity.

The rule of thumb makes it possible to be more precise about what was called ill-conditioned and well-conditioned problems. A problem will be said to be ill-conditioned if the condition number is greater than $1/u$. It means that the relative forward error is greater than 1 just saying that one has no accuracy at all for the computed solution.

In fact, in some cases, the rule of thumb can be proved. For the summation, if one denotes by $\widehat{s}$ the computed sum of the vector $y_i$, $1 \leq i \leq n$ and $s = \sum_{i=1}^{n} y_i$ the real sum, then (3.35) implies

$$\frac{|\widehat{s} - s|}{|s|} \leq \gamma_{n-1} \operatorname{cond}\left(\sum_{i=1}^{n} y_i\right) \tag{3.39}$$

with $\gamma_n$ defined by

$$\gamma_n := \frac{nu}{1 - nu} \quad \text{for } n \in \mathbb{N}. \tag{3.40}$$

Since $\gamma_{n-1} \approx (n-1)u$, it is almost the rule of thumb with just a small factor $n-1$ before $u$.

### 3.1.5 The LAPACK library

The LAPACK library [2] is a collection of subroutines in Fortran 77 designed to solve major problems in linear algebra: linear systems, least square systems, eigenvalues and singular values problems.

One of the most important advantages of LAPACK is that it provides error bounds for all the computed quantities. These error bounds are not rigorous but are mostly reliable. To do this, LAPACK uses the principles of backward analysis. In general, LAPACK provides both componentwise and normwise relative error bounds using the rule of thumb (3.37).

In fact, the major part of the algorithms implemented in LAPACK are backward-stable meaning that the rule of thumb (3.38) is satisfied. As the condition number is generally very hard to compute, LAPACK uses estimators. It may happen that the estimator is far from the right condition number. In fact, the estimation can arbitrarily be far from the true condition number. The error bounds in LAPACK are only qualitative markers of the accuracy of the computed results.

Linear algebra problems are central in current scientific computing. Getting some good error bounds is therefore very important and is still a challenge.

### 3.1.6 The PRECISE toolbox

Let us recall the problem $(P)$: given $y$, find $x$ such that $F(x) = y$. Let us denote by $\widehat{x} = \widehat{G}(y)$ the solution computed in finite precision using algorithm $G$. The principle of PRECISE [6] is to control the solutions corresponding to the following transformations:

$$y \xrightarrow{\widehat{G}} \widehat{x} \xrightarrow{F} \widehat{y} = F(\widehat{x}).$$

The data $z$ of the problem $(P)$ is perturbed by $\Delta z$. The PRECISE method makes random perturbation in the original data and examines the effect on the solution of the problem as follows

$$z + \Delta z \xrightarrow{\widehat{G}} \widehat{x} + \Delta\widehat{x} \xrightarrow{F} \widehat{y} + \Delta\widehat{y} = y + \Delta y$$

Thanks to sampling, it is possible to compute the following estimators

- $\mathcal{I}_\delta = \sup_{\|\Delta z\|=\delta} \frac{\|\Delta y\|}{\delta}$,
- $\mathcal{K}_\delta = \sup_{\|\Delta z\|=\delta} \frac{\|\Delta\widehat{x}\|}{\|\Delta\widehat{y}\|}$,
- $\mathcal{L}_\delta = \sup_{\|\Delta z\|=\delta} \frac{\|\Delta\widehat{x}\|}{\delta}$.

Those estimators can give important informations about the stability of an algorithm.

## 3.2 Interval arithmetic

Interval arithmetic [1, 21] is not defined on real numbers, but on closed bounded intervals. The result of an arithmetic operation between two intervals, $X = [\underline{x}, \overline{x}]$ and $Y = [\underline{y}, \overline{y}]$, contains all values that can be obtained by performing this operation on elements from each interval. The arithmetic operations are defined below.

$$\begin{aligned}
X+Y &= [\underline{x}+\underline{y},\ \overline{x}+\overline{y}]. && (3.41)\\
X-Y &= [\underline{x}-\overline{y},\ \overline{x}-\underline{y}]. && (3.42)\\
X\times Y &= [\min(\underline{x}\times\underline{y},\underline{x}\times\overline{y},\overline{x}\times\underline{y},\overline{x}\times\overline{y}),\ \max(\underline{x}\times\underline{y},\underline{x}\times\overline{y},\overline{x}\times\underline{y},\overline{x}\times\overline{y})]. && (3.43)\\
X^2 &= [\min(\underline{x}^2,\overline{x}^2),\ \max(\underline{x}^2,\overline{x}^2)] \text{ if } 0\notin[\underline{x},\ \overline{x}], && (3.44)\\
&\quad [0,\max(\underline{x}^2,\overline{x}^2)] \text{ otherwise.} &&\\
1/Y &= [\min(1/\underline{y},1/\overline{y}),\ \max(1/\underline{y},1/\overline{y})] \text{ if } 0\notin[\underline{y},\overline{y}]. && (3.45)\\
X/Y &= [\underline{x},\ \overline{x}]\times(1/[\underline{y},\ \overline{y}]) \text{ if } 0\notin[\underline{y},\ \overline{y}]. && (3.46)
\end{aligned}$$

Arithmetic operations can also be applied to interval vectors and interval matrices by performing scalar interval operations componentwise.

An interval extension of a function $f$ must provide all values that can be obtained by applying the function to any element of the interval argument $X$:

$$\forall x\in X,\ f(x)\in f(X). \tag{3.47}$$

For instance, $\exp[\underline{x},\overline{x}]=[\exp\underline{x},\exp\overline{x}]$ and $\sin[\pi/6,2\pi/3]=[1/2,1]$.

The interval obtained may depend on the formula chosen for mathematically equivalent expressions. For instance, let $f_1(x)=x^2-x+1$. $f_1([-2,1])=[-2,7]$. Let $f_2(x)=(x-1/2)^2+3/4$. The function $f_2$ is mathematically equivalent to $f_1$, but $f_2([-2,1])=[3/4,7]\neq f_1([-2,1])$. One can notice that $f_2([-2,1])\subseteq f_1([-2,1])$. Indeed a power set evaluation is always contained in the intervals resulting from other mathematically equivalent formulas.

Interval arithmetic enables one to control rounding errors automatically. On a computer, a real value which is not machine representable can be approximated to a floating-point number. It can also be enclosed by two floating-point numbers. Real numbers can therefore be replaced by intervals with machine representable bounds. An interval operation can be performed using directed rounding modes, in such a way that the rounding error is taken into account and the exact result is necessarily contained in the computed interval. For instance, the computed results, with guaranteed bounds, of the addition and the subtraction between two intervals $X=[\underline{x},\overline{x}]$ and $Y=[\underline{y},\overline{y}]$ are

$$\begin{aligned}
X+Y &= [\nabla(\underline{x}+\underline{y}),\Delta(\overline{x}+\overline{y})]\supseteq\{x+y|x\in X,y\in Y\} && (3.48)\\
X-Y &= [\nabla(\underline{x}-\overline{y}),\Delta(\overline{x}-\underline{y})]\supseteq\{x-y|x\in X,y\in Y\} && (3.49)
\end{aligned}$$

where $\nabla$ (respectively $\Delta$) denotes the downward (respectively upward) rounding mode.

Interval arithmetic has been implemented in several libraries or softwares. For instance, a C++ class library, C-XSC[2] [17], and a Matlab toolbox, INTLAB[3] [15, 36], are freely available.

[2]`http://www.xsc.de`

[3]`http://www.ti3.tu-harburg.de/rump/intlab`

The main advantage of interval arithmetic is its reliability. But the intervals obtained may be too large. The intervals width regularly increases with respect to the intervals that would have been obtained in exact arithmetic. With interval arithmetic, rounding error compensation is not taken into account.

The overestimation of the error can be due to the loss of variable dependency. In interval arithmetic, several occurrences of the same variable are considered as different variables. For instance, let $X = [1, 2]$,

$$\forall x \in X,\ x - x = 0, \tag{3.50}$$

but

$$X - X = [-1, 1]. \tag{3.51}$$

Another source of overestimation is the "wrapping effect" due to the enclosure of a non-interval shape range into an interval. For instance, the image of the square $[0, \sqrt{2}] \times [0, \sqrt{2}]$ by the function

$$f(x, y) = \frac{\sqrt{2}}{2}(x + y, y - x) \tag{3.52}$$

is the rotated square $S_1$ with corners $(0, 0), (1, -1), (2, 0), (1, 1)$. The square $S_2$ provided by interval arithmetic operations is: $f([0, \sqrt{2}], [0, \sqrt{2}]) = ([0, 2], [-1, 1])$. The area obtained with interval arithmetic is twice the one of the rotated square $S_1$.

As the classical numerical algorithms can lead to over-pessimistic results in interval arithmetic, specific algorithms, suited for interval arithmetic, have been proposed. Table 4 presents the results obtained for the determinant of Hilbert matrix $H$ of dimension 8 defined by

$$H_{ij} = \frac{1}{i + j - 1} \quad \text{for} \quad i = 1, ..., 8 \ \text{ and } \quad j = 1, ..., 8 \tag{3.53}$$

computed:

- using the Gaussian elimination in IEEE double precision arithmetic with rounding to nearest
- using the Gaussian elimination in interval arithmetic
- using a specific interval algorithm for the inclusion of the determinant of a matrix, which is described in [33], page 214.

Results obtained in interval arithmetic have been computed using the INTLAB toolbox.

The exact value of the determinant is

$$\det(H) = \prod_{k=0}^{7} \frac{(k!)^3}{(8 + k)!}. \tag{3.54}$$

Its 15 first exact significant digits are:

$$\det(H) = 2.73705011379151\text{E-}33. \tag{3.55}$$

|  | $\det(H)$ | # exact digits |
|---|---|---|
| IEEE double precision | 2.737050300217821E-33 | 7.17 |
| interval Gaussian elimination | [2.717163073713011E-33, 2.756937028322111E-33] | 1.84 |
| interval specific algorithm | [2.737038183754026E-33, 2.737061910503125E-33] | 5.06 |

Table 4: Determinant of the Hilbert matrix $H$ of dimension 8

The number of exact significant decimal digits of each computed result has been reported in Table 4.

One can verify the main feature of interval arithmetic: the exact value of the determinant is enclosed in the computed intervals. Table 4 points out the overestimation of the error with naive implementations of classical numerical algorithms in interval arithmetic. The algorithm for the inclusion of a determinant which is specific to interval arithmetic leads to a much thinner interval. Such interval algorithms exist in most areas of numerical analysis.

Taylor models [4, 25, 27] have been proposed to reduce both the dependency problem and the wrapping effect in interval arithmetic. A Taylor model of a function $f$ on an interval $X$ consists of the Taylor polynomial $p_n$ of order $n$ of $f$ and an interval remainder term $I_n$, which encloses the approximation error $|f - p_n|$ on $X$. In computations that involve $f$, the function is replaced by $p_n + I_n$. The polynomial part is propagated by symbolic calculations where possible. The interval remainder term is processed according to the rules of interval arithmetic. All truncation and rounding errors which occur in intermediate computations are also enclosed into the remainder interval of the final result. Taylor model arithmetic has been implemented in the COSY Infinity package[4] [26].

As a remark, interval analysis can be used not only for reliable numerical simulations but also for computer assisted proofs (cf., for example, [33]).

## 3.3 Probabilistic approach

Here, a method for estimating rounding errors is presented without taking into account the model errors or the discretization errors.

Let us go back to the question *"What is the computing error due to floating-point arithmetic on the results produced by a program?"*. From the physical point of view, in large numerical simulations, the final rounding error is the result of billions and billions elementary rounding errors. In the general case, it is impossible to carefully describe each elementary error and, then, to compute the right value of the final rounding error. It is usual, in physics, when a deterministic approach is not possible, to apply a probabilistic model. Of course, one looses the exact description of the phenomena but one may hope to get some global information like order of magnitude, frequency, etc. It is exactly what is hoped when using

[4]`http://bt.pa.msu.edu/index_cosy.htm`

a probabilistic model of rounding errors.

For the mathematical model, remember the formula at the first order (2.13). Concretely, the rounding mode of the computer is replaced by a random rounding mode, *i.e.* at each elementary operation, the result is rounded towards $-\infty$ or $+\infty$ with the probability 0.5. The main interest of this new rounding mode is to run a same binary code with different rounding error propagations because one generates for different runs different random draws. If rounding errors affect the result, even slightly, one obtains for $N$ different runs, $N$ different results on which a statistical test may be applied. This is the basic idea of the CESTAC method (Contrôle et Estimation STochastique des Arrondis de Calcul). Briefly, the part of the $N$ mantissas that is common to the $N$ results is assumed to be not affected by rounding errors, contrary to the part of the $N$ mantissas which is different from one result to an other.

The implementation of the CESTAC method in a code providing a result $R$ consists in:

- executing $N$ times this code with the random rounding mode, which is obtained by using randomly the rounding mode towards $-\infty$ or $+\infty$; then, an $N$-sample $(R_i)$ of $R$ is obtained,
- choosing as the computed result the mean value $\overline{R}$ of $R_i$, $i = 1, ..., N$,
- estimating the number of exact decimal significant digits of $\overline{R}$ with

  $$C_{\overline{R}} = \log_{10}\left(\frac{\sqrt{N}\left|\overline{R}\right|}{\sigma\tau_\beta}\right), \tag{3.56}$$

  where

  $$\overline{R} = \frac{1}{N}\sum_{i=1}^{N} R_i \quad \text{and} \quad \sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(R_i - \overline{R}\right)^2. \tag{3.57}$$

  $\tau_\beta$ is the value of Student's distribution for $N-1$ degrees of freedom and a probability level $1-\beta$.

From equation (2.13), if the first order approximation is valid, one may deduce that:

1. the mean value of the random variable $R$ is the exact result $r$,
2. under some assumptions, the distribution of $R$ is a quasi-Gaussian distribution.

It has been shown that $N = 3$ is the optimal value. The estimation with $N = 3$ is more reliable than with $N = 2$ and increasing the size of the sample does not improve the quality of the estimation. The complete theory can be found in [8, 37].

Equations (2.13) and (3.56) hold if the following assumptions are verified:

1. the round-off errors $\alpha_i$ are independent, centered uniformly distributed random variables,
2. the approximation to the first order in $2^{-p}$ is legitimate.

Concerning the first assumption, with the use of the random arithmetic, round-off errors $\alpha_i$ are random variables, however, in practice, they are not rigorously centered and in this case Student's test gives a biased estimation of the computed result. It has been proved [10] that, with a bias of a few $\sigma$, the error on the estimation of the number of exact significant digits of $\overline{R}$ is less than one decimal digit. Therefore even if the first assumption is not rigorously satisfied, the reliability of the estimation obtained with equation (3.56) is not altered if it is considered as exact up to one digit.

Concerning the second assumption, the approximation to the first order only concerns multiplications and divisions. Indeed the rounding error generated by an addition or a subtraction does not contain any term of higher order. It has been shown [7, 8] that, if a computed result becomes non significant, *i.e.* all its significant digits are affected by rounding errors, then the first order approximation may be not legitimate. In practice the validation of the CESTAC method requires a dynamical control of multiplications and divisions, during the execution of the code.

This leads to the synchronous implementation of the method, *i.e.* to the parallel computation of the $N$ results $R_i$. In this approach, a classical floating-point number is replaced by a 3-sample $X = (X_1, X_2, X_3)$ and an elementary operation $\Omega \in \{+, -, \times, /\}$ is defined by $X \Omega Y = (X_1 \omega Y_1, X_2 \omega Y_2, X_3 \omega Y_3)$ where $\omega$ represents the corresponding floating-point operation followed by a random rounding.

A new important concept has also been introduced: the computational zero.

**Definition 3.1.** During the run of a code using the CESTAC method, an intermediate or a final result $R$ is a computational zero, denoted by @.0, if one of the two following conditions holds:

- $\forall i, R_i = 0$,
- $C_{\overline{R}} \leq 0$.

Any computed result $R$ is a computational zero if either $R = 0$, $R$ being significant, or $R$ is non significant. In other words, a computational zero is a value that cannot be differentiated from the mathematical zero because of its rounding error. From this new concept of zero, one can deduce new order relationships that take into account the accuracy of intermediate results. For instance,

**Definition 3.2.** $X$ is stochastically strictly greater than $Y$ if and only if:

$$\overline{X} > \overline{Y} \quad \text{and} \quad X - Y \neq @.0.$$

or

**Definition 3.3.** $X$ is stochastically greater than or equal to $Y$ if and only if:

$$\overline{X} \geq \overline{Y} \quad \text{or} \quad X - Y = @.0.$$

The joint use of the CESTAC method and these new definitions is called DSA (Discrete Stochastic Arithmetic) [38]. Elements of the DSA, which are named *stochastic numbers*, are $N$-sets provided by the CESTAC method. DSA enables to estimate the impact of rounding errors on any result of a scientific code and also to check that no anomaly occurred during the run, especially in branching statements. The CADNA (Control of Accuracy and Debugging for Numerical Applications) software[5] [19] is a library which implements DSA in any code written in C++ or in Fortran and allows to use new numerical types: the stochastic types. The library contains the definition of all arithmetic operations and order relations for the stochastic types. The control of the accuracy is only performed on variables of stochastic type. When a stochastic variable is printed, only its exact significant digits appear. For a computational zero, the symbol @.0 is printed. The CADNA library allows, during the execution of any code:

- the estimation of the error due to rounding error propagation,
- the detection of numerical instabilities,
- the checking of the sequencing of the program (tests and branchings),
- the estimation of the accuracy of all intermediate computations.

The benefits of DSA in numerical programs are illustrated by the two following examples. For both examples, at first, results using the standard floating-point arithmetic are presented and, then, results using the CADNA library.

**Example 1: a rational fraction function of two variables**

In the following example [35], the rational fraction

$$F(x,y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + \frac{x}{2y}$$

is computed with $x = 77617$, $y = 33096$.
The 15 first digits of the exact result are -0.827396059946821.
Using IEEE double precision arithmetic with rounding to nearest, one obtains:
res = 5.764607523034235E+17
and using CADNA in double precision, one obtains:

```
----------------------------------------------------
CADNA software --- University P. et M. Curie --- LIP6
Self-validation detection: ON
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
```

[5]http://www.lip6.fr/cadna/

```
 Cancellation instabilities detection: ON
 -----------------------------------------------------
 res = @.0
 -----------------------------------------------------
 CADNA software --- University P. et M. Curie --- LIP6
 There is  1 numerical instability
 0  UNSTABLE DIVISION(S)
 0  UNSTABLE POWER FUNCTION(S)
 0  UNSTABLE MULTIPLICATION(S)
 0  UNSTABLE BRANCHING(S)
 0  UNSTABLE MATHEMATICAL FUNCTION(S)
 0  UNSTABLE INTRINSIC FUNCTION(S)
 1  UNSTABLE CANCELLATION(S)
```

CADNA points out the complete loss of accuracy of the result.

**Example 2: solving a linear system**

In this example, CADNA is able to provide correct results which were impossible to be obtained with the standard floating-point arithmetic. The following linear system is solved using Gaussian elimination with partial pivoting.

$$\begin{pmatrix} 21 & 130 & 0 & 2.1 \\ 13 & 80 & 4.74\ 10^{8} & 752 \\ 0 & -0.4 & 3.9816\ 10^{8} & 4.2 \\ 0 & 0 & 1.7 & 9\ 10^{-9} \end{pmatrix} . X \quad = \quad \begin{pmatrix} 153.1 \\ 849.74 \\ 7.7816 \\ 2.6\ 10^{-8} \end{pmatrix}$$

The exact solution is $x_{sol}^{t} = (1, 1, 10^{-8}, 1)$. Using IEEE single precision arithmetic with rounding to nearest, one obtains:

```
x_sol(1) =     0.6261988E+02     (exact solution:     0.1000000E+01)
x_sol(2) =    -0.8953979E+01     (exact solution:     0.1000000E+01)
x_sol(3) =     0.0000000E+00     (exact solution:     0.1000000E-07)
x_sol(4) =     0.1000000E+01     (exact solution:     0.1000000E+01)
```

and using CADNA in single precision, one obtains:

```
CADNA software --- University P. et M. Curie --- LIP6
Self-validation detection: ON
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
Cancellation instabilities detection: ON
-----------------------------------------------------
x_sol(1) =     0.999E+00         (exact solution:     0.1000000E+01)
x_sol(2) =     0.1000E+01        (exact solution:     0.1000000E+01)
x_sol(3) =     0.999999E-08      (exact solution:     0.9999999E-08)
x_sol(4) =     0.1000000E+01     (exact solution:     0.1000000E+01)
-----------------------------------------------------
CADNA software --- University P. et M. Curie --- LIP6
There are 3 numerical instabilities
0 UNSTABLE DIVISION(S)
0 UNSTABLE POWER FUNCTION(S)
0 UNSTABLE MULTIPLICATION(S)
1 UNSTABLE BRANCHING(S)
0 UNSTABLE MATHEMATICAL FUNCTION(S)
1 UNSTABLE INTRINSIC FUNCTION(S)
1 UNSTABLE CANCELLATION(S)
```

During the reduction of the third column, the matrix element a(3,3) is equal to 4864. But the exact value of a(3,3) is zero. The standard floating-point arithmetic cannot detect that a(3,3) is non-significant. This value is chosen as pivot. That leads to erroneous results. CADNA detects the non-significant value of a(3,3). This value is eliminated as pivot. That leads to satisfactory results.

## 4 Methods for accurate computations

In this section, different methods to increase the accuracy of the computed result of an algorithm are presented. Far from being exhaustive, two classes of methods will be presented. The first class is the class of compensated methods. These methods consist in estimating the rounding error and then adding it to the computed result. The second class are algorithms using multiprecision arithmetic.

### 4.1 Compensated methods

Throughout this subsection, one assumes that the floating-point arithmetic adhers to IEEE 754 floating-point standard in rounding to nearest. One also assume that no overflow nor underflow occurs. The material presented in this section heavily relies on [28].

#### 4.1.1 Error-free transformations (EFT)

One can notice that $a \circ b \in \mathbb{R}$ and $a \circledcirc b \in \mathbb{F}$ but in general $a \circ b \in \mathbb{F}$ does not hold. It is known that for the basic operations $+, -, \times$, the approximation error of a floating-point operation is still a floating-point number:

$$
\begin{array}{llll}
x = a \oplus b & \Rightarrow & a + b = x + y & \text{with } y \in \mathbb{F}, \\
x = a \ominus b & \Rightarrow & a - b = x + y & \text{with } y \in \mathbb{F}, \\
x = a \otimes b & \Rightarrow & a \times b = x + y & \text{with } y \in \mathbb{F}, \\
x = a \oslash b & \Rightarrow & a = x \times b + y & \text{with } y \in \mathbb{F}, \\
x = \bigcirc\!\!\!\!\sqrt{\ }(a) & \Rightarrow & a = x^2 + y & \text{with } y \in \mathbb{F}.
\end{array}
\tag{4.58}
$$

These are *error-free* transformations of the pair $(a, b)$ into the pair $(x, y)$. The floating-point number $x$ is the result of the floating-point operation whereas $y$ is the rounding term. Fortunately, the quantities $x$ and $y$ in (4.58) can be computed exactly in floating-point arithmetic. For the algorithms, Matlab-like notations are used. For addition, one can use the following algorithm by Knuth.

**Algorithm 4.1** (Knuth [20]). Error-free transformation of the sum of two floating-point numbers

```
function $[x, y]$ = TwoSum$(a, b)$
  $x = a \oplus b$
  $z = x \ominus a$
  $y = (a \ominus (x \ominus z)) \oplus (b \ominus z)$
```

Another algorithm to compute an error-free transformation is the following algorithm from Dekker. The drawback of this algorithm is that $x + y = a + b$ provided that $|a| \geq |b|$. Generally, on modern computers, a comparison followed by a branching and 3 operations costs more than 6 operations. As a consequence, `TwoSum` is generally more efficient than `FastTwoSum` plus a branching.

**Algorithm 4.2** (Dekker [11]). Error-free transformation of the sum of two floating-point numbers.

```
function $[x, y]$ = FastTwoSum$(a, b)$
  $x = a \oplus b$
  $y = (a \ominus x) \oplus b$
```

For the error-free transformation of a product, one first needs to split the input argument into two parts. Let $p$ be given by $u = 2^{-p}$ and let us define $s = \lceil p/2 \rceil$. For example, if the working precision is IEEE 754 double precision, then $p = 53$ and $s = 27$. The following algorithm by Dekker splits a floating-point number $a \in \mathbb{F}$ into two parts $x$ and $y$ such that

$$a = x + y \quad \text{with } |y| \leq |x|. \tag{4.59}$$

Both parts $x$ and $y$ have at most $s - 1$ non-zero bits.

**Algorithm 4.3** (Dekker [11]). Error-free split of a floating-point number into two parts

```
function $[x, y]$ = Split$(a, b)$
    factor $= 2^s \oplus 1$
    $c =$ factor $\otimes a$
    $x = c \ominus (c \ominus a)$
    $y = a \ominus x$
```

The main point of `Split` is that both parts can be multiplied in the same precision without error. With this function, an algorithm attributed to Veltkamp by Dekker enables to compute an error-free transformation for the product of two floating-point numbers. This algorithm returns two floating-point numbers $x$ and $y$ such that

$$a \times b = x + y \quad \text{with } x = a \otimes b. \tag{4.60}$$

**Algorithm 4.4** (Veltkamp [11])**.** Error-free transformation of the product of two floating-point numbers

```
function $[x, y]$ = TwoProduct$(a, b)$
    $x = a \otimes b$
    $[a_1, a_2]$ = Split$(a)$
    $[b_1, b_2]$ = Split$(b)$
    $y = a_2 \otimes b_2 \ominus (((x \ominus a_1 \otimes b_1) \ominus a_2 \otimes b_1) \ominus a_1 \otimes b_2)$
```

The performance of the algorithms is interpreted in terms of floating-point operations (flops). The following theorem summarizes the properties of algorithms `TwoSum` and `TwoProduct`.

**Theorem 4.1.** *Let $a, b \in \mathbb{F}$ and let $x, y \in \mathbb{F}$ such that $[x, y] =$ `TwoSum`$(a, b)$ (Algorithm 4.1). Then,*

$$a + b = x + y, \quad x = a \oplus b, \quad |y| \le u|x|, \quad |y| \le u|a + b|. \tag{4.61}$$

*The algorithm* `TwoSum` *requires* 6 *flops.*
*Let $a, b \in \mathbb{F}$ and let $x, y \in \mathbb{F}$ such that $[x, y] =$ `TwoProduct`$(a, b)$ (Algorithm 4.4). Then,*

$$a \times b = x + y, \quad x = a \otimes b, \quad |y| \le u|x|, \quad |y| \le u|a \times b|. \tag{4.62}$$

*The algorithm* `TwoProduct` *requires* 17 *flops.*

The `TwoProduct` algorithm can be re-written in a very simple way if a Fused-Multiply-and-Add (`FMA`) operator is available on the target architecture. Some computers have a *Fused-Multiply-and-Add* (`FMA`) operation that enables a floating-point multiplication followed by an addition to be performed as a single floating-point operation. The Intel IA-64 architecture, implemented in the Intel Itanium processor, has an `FMA` instruction as well as the IBM RS/6000 and the PowerPC before it. On the Itanium processor, the `FMA` instruction enables a multiplication and an addition to be performed in the same number of cycles than one multiplication or one addition. As a result, it seems to be advantageous for speed as well as for accuracy.

Theoretically, this means that for $a, b, c \in \mathbb{F}$, the result of $\texttt{FMA}(a, b, c)$ is the nearest floating-point number of $a \times b + c \in \mathbb{R}$. The `FMA` satisfies

$$\texttt{FMA}(a, b, c) = (a \times b + c)(1 + \varepsilon_1) = (a \times b + c)/(1 + \varepsilon_2) \text{ with } |\varepsilon_\nu| \leq u.$$

Thanks to the `FMA`, the `TwoProduct` algorithm can be re-written as follows which costs only 2 flops.

**Algorithm 4.5.** Error-free transformation of the product of two floating-point numbers using an `FMA`.

```
function $[x, y] = \texttt{TwoProductFMA}(a, b)$
  $x = a \otimes b$
  $y = \texttt{FMA}(a, b, -x)$
```

### 4.1.2 A compensated summation algorithm

Hereafter, a compensated scheme to evaluate the sum of floating-point numbers is presented, *i.e.* the error of individual summation is somehow corrected.

Indeed, with Algorithm 4.1 (`TwoSum`), one can compute the rounding error. This algorithm can be cascaded and sum up the errors to the ordinary computed summation. For a summary, see Figure 3 and Algorithm 4.6.

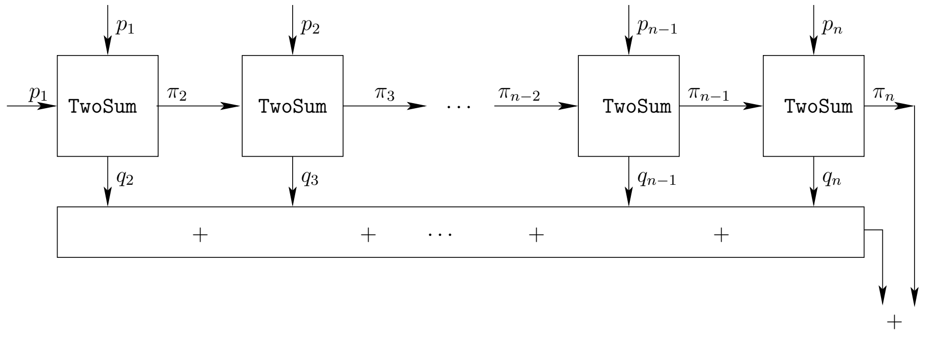

Figure 3: Compensated summation algorithm

**Algorithm 4.6.** Compensated summation algorithm

```
function $\texttt{res} = \texttt{CompSum}(p)$
  $\pi_1 = p_1$ ; $\sigma_1 = 0$;
  for $i = 2 : n$
    $[\pi_i, q_i] = \texttt{TwoSum}(\pi_{i-1}, p_i)$
    $\sigma_i = \sigma_{i-1} \oplus q_i$
  $\texttt{res} = \pi_n \oplus \sigma_n$
```

The following proposition gives a bound on the accuracy of the result. The notation $\gamma_n$ defined by Equation (3.40) will be used. When using $\gamma_n$, $nu \leq 1$ is implicitly assumed.

**Proposition 4.2** ([28])**.** *Suppose Algorithm* `CompSum` *is applied to floating-point numbers $p_i \in \mathbb{F}$, $1 \leq i \leq n$. Let $s := \sum p_i$, $S := \sum |p_i|$ and $nu < 1$. Then, one has*

$$|\texttt{res} - s| \leq u|s| + \gamma_{n-1}^2 S. \tag{4.63}$$

In fact, the assertions of Proposition 4.2 are also valid in the presence of underflow.

One can interpret Equation (4.63) in terms of the condition number for the summation (3.31). Since

$$\operatorname{cond}\left(\sum p_i\right) = \frac{\sum |p_i|}{|\sum p_i|} = \frac{S}{|s|}, \tag{4.64}$$

inserting this in Equation (4.63) yields

$$\frac{|\texttt{res} - s|}{|s|} \leq u + \gamma_{n-1}^2 \operatorname{cond}\left(\sum p_i\right). \tag{4.65}$$

Basically, the bound for the relative error of the result is essentially $(nu)^2$ times the condition number plus the rounding $u$ due to the working precision. The second term on the right hand side reflects the computation in twice the working precision ($u^2$) thanks to the *rule of thumb*. The first term reflects the rounding back in the working precision.

The compensated summation on ill-conditioned sum was tested; the condition number varying from $10^4$ to $10^{40}$.

Figure 4 shows the relative accuracy $|\texttt{res} - s|/|s|$ of the computed value by the two algorithms 3.1 and 4.6. The *a priori* error estimations (3.39) and (4.65) are also plotted.

As one can see in Figure 4, the compensated summation algorithm exhibits the expected behavior, that is to say, the compensated rule of thumb (4.65). As long as the condition number is less than $u^{-1}$, the compensated summation algorithm produces results with full precision (forward relative error of the order of $u$). For condition numbers greater than $u^{-1}$, the accuracy decreases and there is no accuracy at all for condition numbers greater than $u^{-2}$.

Thanks to the `TwoProduct` algorithm, we have already an error-free transformation of a product of two floating-point numbers into the sum of two floating-point numbers. Combining `TwoProduct` with the compensated algorithm `CompSum`, we can provide a compensated algorithm for computing the dot product of two floating-point vectors $x$, $y$ of length $n$.

**Algorithm 4.7.** Compensated dot product algorithm

```
function res = CompDot($x, y$)
  for $i = 1 : n$
    $[r_i, r_{n+i}]$ = TwoProduct($x_i, y_i$)
  end
  res = CompSum($r$)
```

We have similar error bounds as (4.65), see [28] for more details. There exists some improvements for the previous compensated dot product algorithm (see also [28] and the next paragraph).

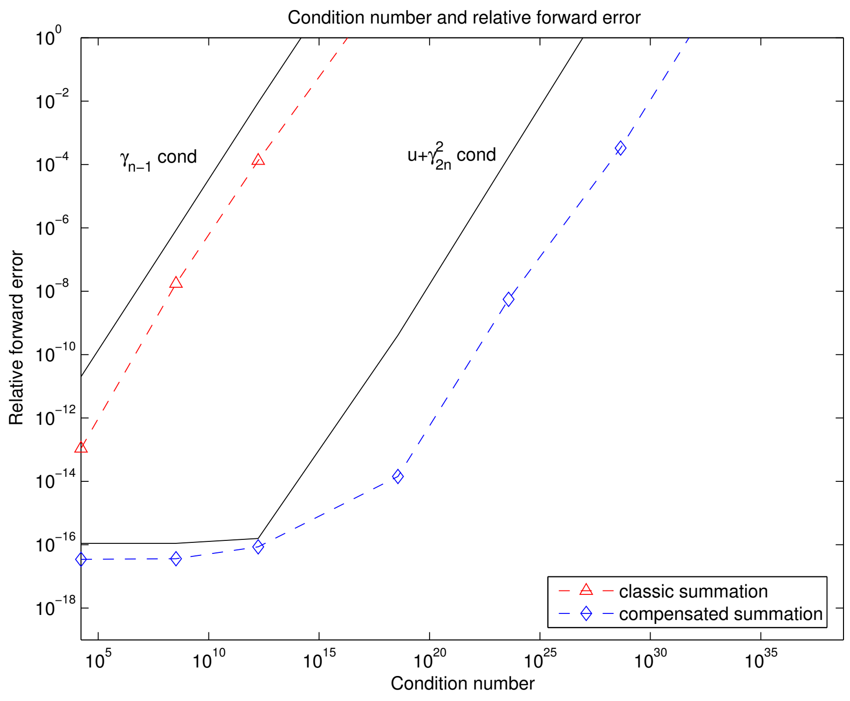

Figure 4: Compensated summation algorithm

### 4.1.3 A compensated dot product algorithm

In the previous paragraph, we have transformed a dot product into a sum and then applied an accurate summation algorithm. In fact, we can do a similar analysis as for the sum. For each elementary operation (here addition and subtraction), we can compute the rounding error thanks to error-free transformations. We can then accumulate the errors and add them to the final result. This is what is done in the following algorithm. We recall that given two vectors of size $n$ of floating-point numbers, the dot product is defined by

$$x^T y = \sum_{i=1}^{n} x_i y_i.$$

**Algorithm 4.8** ([28]). Dot product in twice the working precision

```
function $\texttt{res} = \texttt{CompDot2}(x, y)$
  $[p, s] = \texttt{TwoProduct}(x_1, y_1)$
  for $i = 2 : n$
    $[h, r] = \texttt{TwoProduct}(x_i, y_i)$
    $[p, q] = \texttt{TwoSum}(p, h)$
    $s = \mathrm{fl}(s + (q + r))$
  end
  $\texttt{res} = \mathrm{fl}(p + s)$
```

The following theorem gives an *a priori* error bound on the error.

**Proposition 4.3** ([28]). *Let floating-point numbers $x_i, y_i \in \mathbb{F}, 1 \le i \le n$, be given and denote by* $\texttt{res} \in \mathbb{F}$ *the result computed by Algorithm* CompDot2. *Then*

$$|\texttt{res} - x^T y| \le u|x^T y| + \gamma_n^2 |x^T||y|. \tag{4.66}$$

One can interpret Equation (4.66) in terms of the condition number for dot product. Since

$$\mathrm{cond}\left(x^T y\right) = \frac{|x|^T |y|}{|x^T y|}, \tag{4.67}$$

inserting this in Equation (4.66) yields

$$\frac{|\texttt{res} - x^T y|}{|x^T y|} \le u + \gamma_n^2 \, \mathrm{cond}\left(x^T y\right). \tag{4.68}$$

As a consequence, we conclude that the result is as accurate as if computed in twice the working precision and then rounded to the current working precision. This is the same phenomenon as for the compensated summation algorithm.

### 4.1.4 A compensated Horner scheme

We present hereafter a compensated algorithm for Horner scheme. One can find a more detailed description of the algorithm in [14]. We first recall the classic algorithm for Horner scheme and give an error bound. We then present the compensated Horner scheme together with an error bound.

The classical method for evaluating a polynomial

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

is the Horner scheme which consists in the following algorithm.

**Algorithm 4.9.** Polynomial evaluation with Horner scheme

```
function $\texttt{res} = \texttt{Horner}(p, x)$
$s_n = a_n$
for $i = n-1 : -1 : 0$
  $s_i = s_{i+1} \otimes x \oplus a_i$
end
$\texttt{res} = s_0$
```

A forward error bound for the result of Algorithm 4.9 is (see [16, p.95]):

$$|p(x) - \texttt{res}| \leq \gamma_{2n} \sum_{i=0}^{n} |a_i||x|^i = \gamma_{2n}\widetilde{p}(|x|)$$

where $\widetilde{p}(x) = \sum_{i=0}^{n} |a_i|x^i$. It is very interesting to express and interpret this result in terms of the condition number of the polynomial evaluation defined by

$$\operatorname{cond}(p, x) = \frac{\sum_{i=0}^{n} |a_i||x|^i}{|p(x)|} = \frac{\widetilde{p}(|x|)}{|p(x)|}. \tag{4.69}$$

Thus we have

$$\frac{|p(x) - \texttt{res}|}{|p(x)|} \leq \gamma_{2n} \operatorname{cond}(p, x).$$

If an FMA instruction is available, then the statement $s_i = s_{i+1} \otimes x \oplus a_i$ in Algorithm 4.9 can be re-written as $s_i = \texttt{FMA}(s_{i+1}, x, a_i)$ which slightly improves the error bound. Using an FMA this way, the computed result now satisfies

$$\frac{|p(x) - \texttt{res}|}{|p(x)|} \leq \gamma_n \operatorname{cond}(p, x).$$

We can modify the Horner scheme to compute the rounding error at each elementary operation that is an addition or a multiplication. This is done in Algorithm 4.10.

**Algorithm 4.10.** Polynomial evaluation with a compensated Horner scheme

```
function res = CompHorner($p, x$)
$s_n = a_n$
$r_n = 0$
for $i = n-1 : -1 : 0$
    $[p_i, \pi_i]$ = TwoProduct($s_{i+1}, x$)
    $[s_i, \sigma_i]$ = TwoSum($p_i, a_i$)
    $r_i = r_{i+1} \otimes x \oplus (\pi_i \oplus \sigma_i)$
end
res = $s_0 \oplus r_0$
```

If we denote by $p_\pi$ and $p_\sigma$ the two following polynomials

$$p_\pi = \sum_{i=0}^{n-1} \pi_i x^i, \qquad p_\sigma = \sum_{i=0}^{n-1} \sigma_i x^i,$$

then one can show due to error-free transformations that

$$p(x) = s_0 + p_\pi(x) + p_\sigma(x).$$

If one look closely at the previous algorithm, it is then clear that $s_0 = \texttt{Horner}(p, x)$. As a consequence, we can derive a new error-free transformation for polynomial evaluation

$$p(x) = \texttt{Horner}(p, x) + p_\pi(x) + p_\sigma(x).$$

The compensated Horner scheme first computes $p_\pi(x) + p_\sigma(x)$ which corresponds to the rounding errors and then add it to the result of the classic Horner scheme $\texttt{Horner}(p, x)$.

We will show that the results computed according to Algorithm 4.10 admit significantly better error-bounds than those computed with the classical Horner scheme. We argue that Algorithm 4.10 provides results as if they were computed using twice the working precision. This is summed up in the following theorem.

**Theorem 4.4.** *Consider a polynomial $p$ of degree $n$ with floating-point coefficients, and $x$ a floating-point value. The forward error in the compensated Horner algorithm is such that*

$$|\texttt{CompHorner}(p, x) - p(x)| \le u|p(x)| + \gamma_{2n}^2 \widetilde{p}(x). \tag{4.70}$$

It is interesting to interpret the previous theorem in terms of the condition number of the polynomial evaluation of $p$ at $x$. Combining the error bound (4.70) with the condition number (4.69) for polynomial evaluation gives

$$\frac{|\texttt{CompHorner}(p, x) - p(x)|}{|p(x)|} \le u + \gamma_{2n}^2 \operatorname{cond}(p, x). \tag{4.71}$$

In other words, the bound for the relative error of the computed result is essentially $\gamma_{2n}^2$ times the condition number of the polynomial evaluation, plus the inevitable $u$ term for rounding the result to the working precision. In particular, if $\mathrm{cond}(p, x) < \gamma_{2n}^{-1}$, then the relative accuracy of the result is bounded by a constant of the order $u$. This means that the compensated Horner algorithm computes an evaluation accurate to the last few bits as long as the condition number is smaller than $\gamma_{2n}^{-1} \approx (2nu)^{-1}$. Besides that, (4.71) tells us that the computed result is as accurate as if computed by the classic Horner algorithm with twice the working precision, and then rounded to the working precision.

We test the expanded form of the polynomial $p_n(x) = (x-1)^n$. The argument $x$ is chosen near the unique real root 1 of $p_n$, and with many significant bits so that a lot of rounding errors occur during the evaluation of $p_n(x)$. We increment the degree $n$ from 1 until a sufficiently large range has been covered by the condition number $\mathrm{cond}(p_n, x)$. Here we have

$$\mathrm{cond}(p_n, x) = \frac{\widehat{p_n}(x)}{|p_n(x)|} = \left|\frac{1+x}{1-x}\right|^n,$$

and $\mathrm{cond}(p_n, x)$ grows exponentially with respect to $n$. In the experiments reported in Figure 5, $\mathrm{cond}(p_n, x)$ varies from $10^2$ to $10^{40}$ (for $x = \mathrm{fl}(1.333)$, that corresponds to the degree range $n = 3, \ldots, 42$). These huge condition numbers have a sense since here the coefficients of $p$ and the value $x$ are chosen to be exact floating-point numbers.

We experiment both `Horner` and `CompHorner`. For each polynomial $p_n$, the exact value $p_n(x)$ is approximated with a high accuracy. Figure 5 presents the relative accuracy $|y - p_n(x)|/|p_n(x)|$ of the evaluation $y$ computed by the two algorithms.

We observe that the compensated algorithm exhibits the expected behavior. The full precision solution is computed as long as the condition number is smaller than $u^{-1} \approx 10^{16}$. Then, for condition numbers between $u^{-1}$ and $u^{-2} \approx 10^{32}$, the relative error degrades to no accuracy at all. However, when the condition number is beyond $u^{-1}$, the *a priori* error estimate 4.71 is always pessimistic by 2 or 3 orders of magnitude.

### 4.1.5 The CENA method

We now present an automatic correction method [22]. Assume that one must evaluate a function $f$ at the data $D = (d_1, \ldots, d_n)$ with an algorithm $g$ using intermediate variables $d_{n+1}, \ldots, d_N$. We assume for simplicity that $f$ and $d_i$ are scalar quantities. We also assume that each $d_k$ $(k > n)$ is the result of an elementary operation $+, -, \times, /$ or a square root. We denote by $\delta_k$ the rounding error associated to that operation. To the first order, the computed solution $\widehat{d_N}$ satisfies

$$\widehat{d_N} - f(d_1, \ldots, d_n) = \sum_{k=n+1}^{N} \frac{\partial g}{\partial d_k}(D, \delta)\delta_k. \tag{4.72}$$

The CENA method [22] computes a linearization (4.72) of the rounding error and adds this quantity to the computed solution. The derivatives are computed with automatic differentiation tools and the elementary rounding errors are computed exactly with the error-free transformations. For a certain class of algorithms called *linear* in [22] (classic algorithms

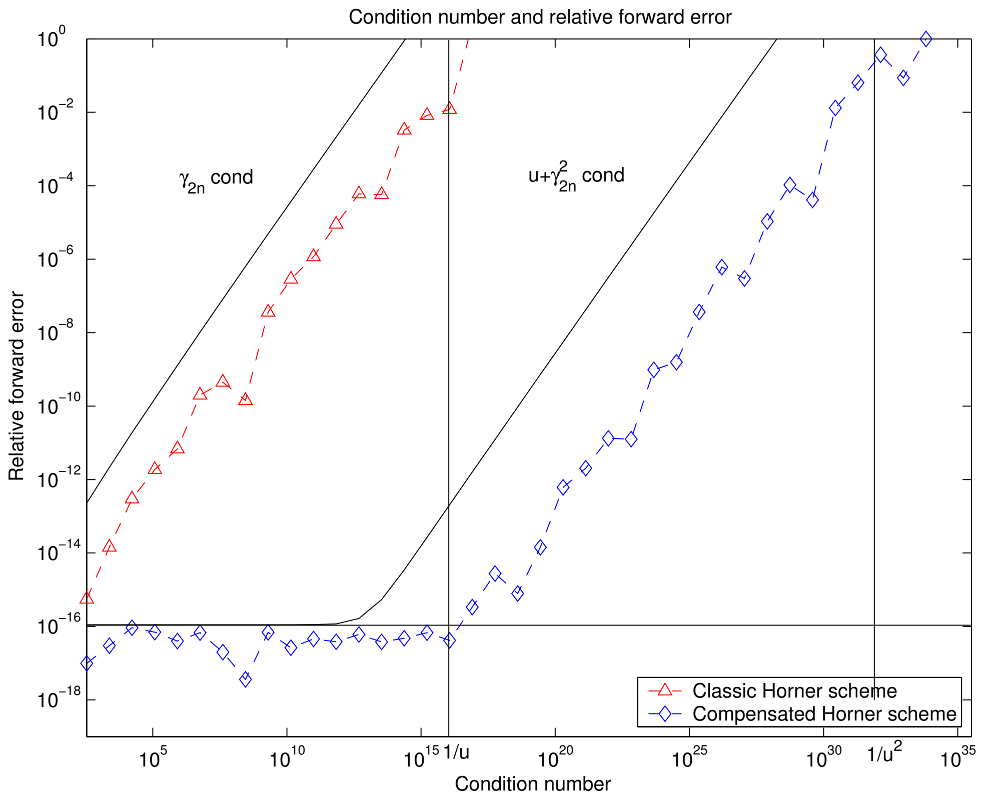

Figure 5: Comparison of the classic Horner scheme with the Compensated Horner scheme

for summation and dot product as well as the Horner scheme are linear algorithms in the sense of [22]), the relation (4.72) is exact and the CENA method returns a solution with a good accuracy (generally as accurate as if computed in twice the working precision and then rounded to the working precision).

### 4.1.6 $K$-fold, faithfully rounded and rounded to nearest results

In the previous sections, we have presented some compensated algorithms that make it possible to compute a result as accurate as if computed with twice the working precision and then rounded to the current working precision.

Let us generalize this for a general problem. Let $\widehat{x}$ be the computed solution of a problem $(P)$ whose exact solution is $x$. Suppose that the computations have been done with floating-point arithmetic with unit round-off $u$. The computed solution $\widehat{x}$ is as accurate as if computed with twice the working precision if

$$\frac{|\widehat{x}-x|}{|x|} \le u + Cu^2 \operatorname{cond}(P), \tag{4.73}$$

where $C$ is a moderate constant, $|\cdot|$ is a norm on the space of the solutions and $\operatorname{cond}(P)$ is the condition number of the problem $(P)$. In the right-hand side of inequality (4.73), the second term reflects the computation in twice the working precision and the first one the rounding into the working precision. Relation (4.73) is what we called the compensated rule of thumb, the classic rule of thumb being [16, p.9]

$$\frac{|\widehat{x}-x|}{|x|} \le Cu \operatorname{cond}(P).$$

We will say that the computed result $\widehat{x}$ is of the same quality as if computed in $K$-fold precision and rounded to working precision if

$$\frac{|\widehat{x}-x|}{|x|} \le u + (Cu)^K \operatorname{cond}(P).$$

One can find some $K$-fold precision algorithms for summation and dot product in [28].

Sometimes, it is needed to get even more accuracy. The floating-point predecessor and successor of a real number $r$ satisfying $\min\{f : f \in \mathbb{R}\} < r < \max\{f : f \in \mathbb{F}\}$ are defined by

$$\operatorname{pred}(r) := \max\{f \in \mathbb{F} : f < r\} \quad \text{and} \quad \operatorname{succ}(r) := \min\{f \in \mathbb{F} : r < f\}.$$

**Definition 4.1.** A floating-point number $f \in \mathbb{F}$ is called a *faithful rounding* of a real number $r \in \mathbb{R}$ if

$$\operatorname{pred}(f) < r < \operatorname{succ}(f).$$

We denote this by $f \in \square(r)$. For $r \in \mathbb{F}$, this implies that $f = r$.

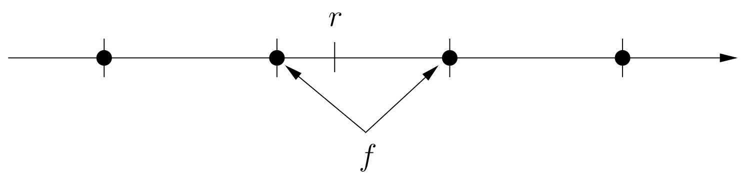

Figure 6: Faithful rounding

Faithful rounding means that the computed result is equal to the exact result if the latter is a floating-point number and otherwise is one of the two floating-point numbers adjacent to the exact result (see Figure 6).

We can be more accurate requiring the rounded to nearest number. As already defined in 2.2, a floating-point number $f \in \mathbb{F}$ is called a *rounded to nearest* of a real number $r \in \mathbb{R}$ if

$$|r - f| = \min\{|r - f'| : f' \in \mathbb{F}\}.$$

The tie can be rounded in any way, for exemple to even.

Recently, some new algorithms for summation and dot product have been proposed [30, 31]. They make it possible to compute a faithful rounding or a rounding to nearest.

## 4.2 Multiple precision arithmetic

Compensated methods are a possible way to improve accuracy, but they force to modify the algorithms. Another possibility is to increase the working precision. For this purpose, some multiprecision libraries have been developed. One can divide the libraries into four categories.

#### Arbitrary precision libraries using a ***multiple-digit*** format

In these libraries a number is expressed as a sequence of digits coupled with a single exponent. Examples of this format are Bailey's ARPREC[6] [3], Brent's MP[7] [5] or MPFR[8] [12]. ARPREC and MPFR are briefly described below. MP is obsolescent: very few changes to the code or documentation have been made since 1981.

The ARPREC library, which is entirely written in C++, supports high-precision real, integer and complex datatypes. Both C++ and Fortran-90 translation modules modules are also provided that permit one to convert an existing C++ or Fortran-90 program to use the library with only minor changes to the source code. In most cases only the type statements and (in the case of Fortran-90 programs) read/write statements need be changed.

[6]`http://crd.lbl.gov/~dhbailey/mpdist/`

[7]`http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub043.html` or `http://wwwmaths.anu.edu.au/~brent/pub/pub043.html`

[8]`http://www.mpfr.org/`

The ARPREC package also includes "The Experimental Mathematician's Toolkit", which is a complete interactive high-precision arithmetic computing environment. One enters expressions in a Mathematica-style syntax, and the operations are performed using the ARPREC package, with a level of precision that can be set from 100 to 1000 decimal digit accuracy. This program supports all basic arithmetic operations, common transcendental and combinatorial functions, high-precision quadrature, summation of series,...

The MPFR library is written in C language and is based on the GNU MP library (GMP for short). The internal representation of a floating-point number $x$ by MPFR is

- a mantissa $m$;
- a sign $s$;
- a signed exponent $e$.

If the precision of $x$ is $p$, then the mantissa $m$ has $p$ significant bits. The mantissa $m$ is represented by an array of GMP unsigned machine-integer type and is interpreted as $1/2 \leq m < 1$. As a consequence, MPFR does not allow denormalized numbers.

MPFR provides the four IEEE rounding modes as well as some elementary functions (*e.g.* exp, log, cos, sin), all correctly rounded. The semantic in MPFR is as follows: for each instruction $a = b + c$ or $a = f(b, c)$ the variables may have different precisions. In MPFR, the data $b$ and $c$ are considered with their full precision and a correct rounding to the full precision of $a$ is computed.

Applications using MPFR inherit the same properties as programs using the IEEE 754 standard (portability, well-defined semantics, possibility to design robust programs and prove their correctness) with no significant slowdown on average with respect to multiple precision libraries with ill-defined semantics.

**Arbitrary precision libraries using a *multiple-component* format**

In these libraries a number is expressed as unevaluated sums of ordinary floating-point words. Examples using this format are Priest's[9] and Shewchuk's[10] libraries. Such a format is also introduced in [34]. An expansion $x$ is then a unevaluated sums of several floating-point numbers

$$x = x_1 + x_2 + \cdots + x_m.$$

Each $x_i$ is called a component of $x$ and is a floating-point number. To impose some structure on the expansion, the numbers $x_i$ are required to be non-overlapping and ordered by magnitude ($|x_1| \leq |x_2| \leq \cdots \leq |x_m|$). Priest says that $x$ and $y$ non-overlap (with $|x| \leq |y|$) if the most significant bit of $x$ is lowest than the least significant bit of $y$. This is the main difference with Shewchuck's expansions where $x$ and $y$ non-overlap if the most significant nonzero bit of $x$ is less than the least significant nonzero bit of $y$. Note that the number $m$ of floating-point numbers is not fixed. For exemple if $x = x_1 + x_2 + \cdots + x_m$ and $y = x_1 + x_2 + \cdots + x_n$ then the product $xy$ is a expansion that can have up to $nm$ terms.

[9] `ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z`

[10] `http://www.cs.cmu.edu/~quake/robust.html`

## Fixed precision libraries using a *multiple-component* format

These libraries use the *multiple-component* format but with a limited number of components. Examples of this format are Bailey's *double-double*[6] (double-double numbers are represented as an unevaluated sum of a leading double and a trailing double) and *quad-double*[6].

The double-double library will be now presented. For our purpose, it suffices to know that a double-double number $a$ is the pair $(a_h, a_l)$ of IEEE-754 floating-point numbers with $a = a_h + a_l$ and $|a_l| \leq u|a_h|$. In the sequel, algorithms for

- the addition of a double number to a double-double number;
- the product of a double-double number by a double number;
- the addition of a double-double number to a double-double number

will only be presented. Of course, it is also possible to implement the product of a double-double by a double-double as well as the division of a double-double by a double, etc.

**Algorithm 4.11.** Addition of the double number $b$ to the double-double number $(a_h, a_l)$

```
function $[c_h, c_l]$ = add_dd_d($a_h, a_l, b$)
  $[t_h, t_l]$ = TwoSum($a_h, b$)
  $[c_h, c_l]$ = FastTwoSum($t_h, (t_l \oplus a_l)$)
```

**Algorithm 4.12.** Product of the double-double number $(a_h, a_l)$ by the double number $b$

```
function $[c_h, c_l]$ = prod_dd_d($a_h, a_l, b$)
  $[s_h, s_l]$ = TwoProduct($a_h, b$)
  $[t_h, t_l]$ = FastTwoSum($s_h, (a_l \otimes b)$)
  $[c_h, c_l]$ = FastTwoSum($t_h, (t_l \oplus s_l)$)
```

**Algorithm 4.13.** Addition of the double-double number $(a_h, a_l)$ to the double-double number $(b_h, b_l)$

```
function $[c_h, c_l]$ = add_dd_dd($a_h, a_l, b_h, b_l$)
  $[s_h, s_l]$ = TwoSum($a_h, b_h$)
  $[t_h, t_l]$ = TwoSum($a_l, b_l$)
  $[t_h, s_l]$ = FastTwoSum($s_h, (s_l \oplus t_h)$)
  $[c_h, c_l]$ = FastTwoSum($t_h, (t_l \oplus s_l)$)
```

Algorithms 4.11 to 4.13 use error-free transformations and are very similar to compensated algorithms. The difference lies in the step of renormalization. This step is the last one in each algorithm and makes it possible to ensure that $|c_l| \leq u|c_h|$.

There are several implementations for the double-double library. The difference is that the lower-order terms are treated in a different way. If $a, b$ are double-double numbers and $\circledcirc \in \{+, \times\}$, then one can show [24] that

$$\mathrm{fl}(a \circledcirc b) = (1+\delta)(a \circledcirc b),$$

with $|\delta| \leq 4 \cdot 2^{-106}$.

One might also note that, when keeping $[\pi_n, \sigma_n]$ as a pair, the first summand $u$ disappears in (4.65) (see [28]), so this is an example for a double-double result.

### The MPFI library for multiprecision interval arithmetic

MPFI (Multiple Precision Floating-point Interval)[11] [29] is intended to be a portable library written in C for arbitrary precision interval arithmetic with intervals represented using MPFR reliable floating-point numbers. It is based on the GNU MP library and on the MPFR library. The purpose of an arbitrary precision interval arithmetic is on the one hand to get guaranteed results, thanks to interval computation, and on the other hand to obtain accurate results, thanks to multiple precision arithmetic. The MPFI library is built upon MPFR in order to benefit from the correct roundings provided by MPFR. Further advantages of using MPFR are its portability and compliance with the IEEE 754 standard for floating-point arithmetic.

## References

[1] G. Alefeld and J. Herzberger. *Introduction to interval analysis.* Academic Press, 1983.

[2] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.).* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[3] D. H. Bailey, Y. Hida, X. S. Li, and B. Thompson. ARPREC: An Arbitrary Precision Computation Package. Technical Report LBNL-53651, Lawrence Berkeley National Laboratory, September 2002.

[4] M. Berz. From Taylor series to Taylor models. In *AIP Conference Proceedings 405*, pages 1–23, 1997.

[5] R. P. Brent. Algorithm 524: MP, A Fortran Multiple-Precision Arithmetic Package. *ACM Trans. Math. Softw.*, 4(1):71–81, 1978.

[6] F. Chaitin-Chatelin and V. Frayssé. *Lectures on finite precision computations.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1996.

[7] J.-M. Chesneaux. Study of the computing accuracy by using probabilistic approach. In C. Ullrich, editor, *Contribution to Computer Arithmetic and Self-Validating Numerical Methods*, pages 19–30, IMACS, New Brunswick, New Jersey, USA, 1990.

[8] J.-M. Chesneaux. *L'arithmétique stochastique et le logiciel CADNA*. Habilitation à diriger des recherches, Université Pierre et Marie Curie, Paris, November 1995.

[9] J.-M. Chesneaux, S. Graillat, and F. Jézéquel. Rounding errors. In Benjamin Wah, editor, *Wiley Encyclopedia of Computer Science and Engineering*, volume 4, pages 2480–2494. John Wiley & Sons, Hoboken, N.J., January 2009.

[11] `http://perso.ens-lyon.fr/nathalie.revol/mpfi_toc.html`

[10] J.-M. Chesneaux and J. Vignes. Sur la robustesse de la méthode CESTAC. *C. R. Acad. Sci. Paris Sér. I Math.*, 307:855–860, 1988.

[11] T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18:224–242, 1971.

[12] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13, 2007. `http://www.mpfr.org`.

[13] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[14] S. Graillat, N. Louvet, and P. Langlois. Compensated Horner scheme. Research Report 04, Équipe de recherche DALI, Laboratoire LP2A, Université de Perpignan Via Domitia, France, 52 avenue Paul Alduy, 66860 Perpignan cedex, France, July 2005.

[15] G. I. Hargreaves. Interval analysis in MATLAB. Numerical Analysis Report No. 416, Manchester Centre for Computational Mathematics, University of Manchester, December 2002. Available at `http://www.maths.man.ac.uk/~nareports/narep416.pdf`.

[16] N. J. Higham. *Accuracy and stability of numerical algorithms.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2002.

[17] W. Hofschuster and W. Krämer. C-XSC 2.0: A C++ Library for Extended Scientific Computing. In *Numerical Software with Result Verification, Lecture Notes in Computer Science*, volume 2991/2004, pages 15–35. Springer-Verlag, Heidelberg, 2004.

[18] IEEE Computer Society, New York. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. Reprinted in SIGPLAN Notices, 22(2):9-25, 1987.

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

[20] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms.* Addison-Wesley, Reading, MA, USA, third edition, 1998.

[21] U. W. Kulisch. *Advanced Arithmetic for the Digital Computer.* Springer-Verlag, Wien, 2002.

[22] P. Langlois. Automatic linear correction of rounding errors. *BIT*, 41(3):515–539, 2001.

[23] P. Langlois. Analyse d'erreur en précision finie. In A. Barraud, editor, *Outils d'analyse numérique pour l'Automatique*, Traité IC2, chapter 1, pages 19–52. Hermes Science, 2002.

[24] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, 2002.

[25] K. Makino and M. Berz. Remainder differential algebras and their appplications. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications and Tools*, pages 63–74. SIAM, Philadelphia, 1996.

[26] K. Makino and M. Berz. Cosy infinity version 9. *Nuclear Instruments and Methods*, A558:346–350, 2005.

[27] M. Neher. From Interval Analysis to Taylor Models - An Overview. In *Proc. IMACS 2005*, Paris, France, 2005.

[28] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput.*, 26(6):1955–1988, 2005.

[29] N. Revol and F. Rouillier. Motivations for an Arbitrary Precision Interval Arithmetic and the MPFI Library. *Reliable Computing*, 11(4):275–290, 2005.

[30] S. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM J. Sci. Comput.*, 31(1):189–224, 2008.

[31] S. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part II: Sign, $k$-fold faithful and rounding to nearest. *SIAM J. Sci. Comput.*, 31(2):1269–1302, 2008.

[32] S. M. Rump. How reliable are results of computers? *Jahrbuch Überblicke Mathematik*, pages 163–168, 1983.

[33] S. M. Rump. Computer-assisted proofs and self-validating methods. In B. Einarsson, editor, *Accuracy and Reliability in Scientific Computing*, Software-Environments-Tools, pages 195–240. SIAM, Philadelphia, PA, 2005.

[34] S. M. Rump, T. Ogita, and S. Oishi. Accurate Floating-point Summation II: Sign, K-fold Faithful and Rounding to Nearest. Technical Report 07.2, Faculty for Information and Communication Sciences, Hamburg University of Technology, April 2007.

[35] S.M. Rump. *Reliability in Computing. The Role of Interval Methods in Scientific Computing.* Academic Press, 1988.

[36] S.M. Rump. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999.

[37] J. Vignes. A stochastic arithmetic for reliable scientific computation. *Math. Comput. Simulation*, 35:233–261, 1993.

[38] J. Vignes. Discrete stochastic arithmetic for validating results of numerical software. *Num. Algo.*, 37(1–4):377–390, dec 2004.

[39] J. H. Wilkinson. Rounding errors in algebraic processes. (32), 1963. Also published by Prentice-Hall, Englewood Cliffs, New Jersey, USA. Reprinted by Dover, New York, 1994.