R

ROUNDING ERRORS

INTRODUCTION

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 50 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).

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 they want to use. The decimal fixed-point notation was introduced at the end of the sixteenth century consecutively by S. Stévin (1582, France), J. Bürgi (1592, Switzerland), and G. Magini (1592, Italy). It remains the notation used worldwide today. Although 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 the 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 that is called the rounding error. Scientists have been well aware of this for four centuries. In the nineteenth 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 30% of his paper to error computation. Unfortunately, this use has completely disappeared since the beginning of the 1960s because of the improvement of computers. When eight 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 nonzero 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 (1). At 1600 km/h, 0.34 second corresponds to approximatively 500 meters, the interception failed and 28 people were killed. With the second formulation, whatever the absolute time is, if no overflow occurs for *icount*, then 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 (2).

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?"*

COMPUTER ARITHMETIC

Representation of Numbers

In a numeral system, numbers are represented by a sequence of symbols. The number of distinct symbols that can be used is called the radix (or the base). For instance, in the decimal system, where the radix is 10, the 10 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 $a_{n-1}a_{n-2}\dots a_1a_0$ with

$$X = \sum_{i=0}^{n-1} a_i b^i \quad \text{and} \quad a_i \in \{0, \dots, b-1\}.$$

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 $a_{n-1}a_{n-2}\dots a_1a_0$ with $a_i \in \{0,\dots,b-1\}$ 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 that is encoded on m digits for its magnitude (e.g., its integer part) and f digits for its fractional part is represented by $a_{m-1}\dots a_0 \cdot a_{-1}\dots a_{-f}$, with

$$X = \sum_{i=-f}^{m-1} a_i b^i \quad \text{and} \quad a_i \in \{0,\dots,b-1\}.$$

If $b = 2$, then 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 ε_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

$$M_X = \sum_{i=0}^{p-1} a_i b^{-i} \quad \text{and} \quad a_i \in \{0,\dots,b-1\}.$$

The mantissa M_X can be written as $M_X = a_0 . a_1 \dots a_{p-1}$. 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).

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. Different values could be used for the radix, the length of the exponent, the length of the mantissa, and so on. So, in 1985, the IEEE 754 standard (3) 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 that is actually stored is a biased exponent E_Δ such that $E_\Delta = E + \Delta$, Δ 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. 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 nonbiased exponent is therefore $E_{max} + 1$.

NaN (Not a Number) is a special value that 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 nonzero 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

Table 1. Exponent Coding in Single and Double Precision

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

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:

$$X = \varepsilon_X M_X 2^{E_{min}} \text{ with } \varepsilon_X = \pm 1,$$

$$M_X = \sum_{i=1}^{p-1} a_i 2^{-i}$$

and

$$a_i \in \{0, 1\}.$$

The mantissa M_X can be written as $M_X = 0.a_1 \ldots a_{p-1}$. Therefore, the lowest positive denormalized number is $\underline{u} = 2^{E_{min}+1-p}$. Moreover, denormalized numbers and gradual underflow imply the nice equivalence $a = b \Leftrightarrow a - b = 0$.

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 that is not machine representable. If $x \in (X_{min}, X_{max})$, then $\{X^-, X^+\} \subset \mathbb{F}^2$ exists such as $X^- < x < X^+$ and $(X^-, X^+) \cap \mathbb{F} = \emptyset$. A rounding mode is a rule that, 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 (+, −, ×, /) 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 toward $+\infty$ (or upward rounding), x is represented by X^+,
- rounding toward $-\infty$ (or downward rounding), x is represented by X^-,
- rounding toward 0, if x is negative, then it is represented by X^+, if x is positive, then it is represented by X^-,
- rounding to the nearest, x is represented by its nearest machine number. If x is at the same distance of X^- and X^+, then it is represented by the machine number that 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 in which a number is not representable as a normalized number, but still as a denormalized one.

Rounding Error Formalization

Notion of Exact Significant Digits. To quantify the accuracy of a computed result correctly, 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 that are in common with r:

$$C_{R,r} = \log_{10}\left|\frac{R + r}{2(R - r)}\right| \tag{1}$$

This mathematical definition is in accordance with the intuitive idea of decimal significant digits in common between two numbers. Indeed Equation (1) is equivalent to

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

If $C_{R,r} = 3$, then 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 caused by the sequences of "0" or "9" is illusive. The significant decimal digits of R and r are really different from the sixth position.

Rounding Error that Occurs 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 Equation (1)

$$X = \varepsilon_X M_X 2^{E_X} \tag{3}$$

and

$$X = x - \varepsilon_X 2^{E_X - p} \alpha_X \tag{4}$$

where α_X represents the normalized rounding error.

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

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

$$X = x(1 + \delta) \text{ with } |\delta| \le u \tag{5}$$

where $u = \varepsilon/2$ with rounding to the nearest and $u = \varepsilon$ with the other rounding modes. The model associated with Equation (5) ignores the possibility of underflow. To take underflow into account, one must modify it to

$$X = x(1 + \delta) + \eta \text{ with } |\delta| \le u \tag{6}$$

and $|\eta| \leq \underline{u}/2$ with rounding to the 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)

$$X_i = x_i - \varepsilon_i 2^{E_i - p}\alpha_i \quad \text{for} \quad i = 1, 2 \tag{7}$$

The errors caused by arithmetic operations that have X_1 and X_2 as operands are given below. For each operation, let us denote by E_3 and ε_3 the exponent and the sign of the computed result. α_3 represents the rounding error performed on the result. Let us denote by $\oplus$, $\ominus$, $\otimes$, $\oslash$ the arithmetic operators on a computer.

$$X_1 \oplus X_2 = x_1 + x_2 - \varepsilon_1 2^{E_1 - p}\alpha_1 - \varepsilon_2 2^{E_2 - p}\alpha_2 - \varepsilon_3 2^{E_3 - p}\alpha_3 \tag{8}$$

Similarly

$$X_1 \ominus X_2 = x_1 - x_2 - \varepsilon_1 2^{E_1 - p}\alpha_1 + \varepsilon_2 2^{E_2 - p}\alpha_2 - \varepsilon_3 2^{E_3 - p}\alpha_3 \tag{9}$$

$$X_1 \otimes X_2 = x_1 x_2 - \varepsilon_1 2^{E_1 - p}\alpha_1 x_2 - \varepsilon_2 2^{E_2 - p}\alpha_2 x_1 + \varepsilon_1\varepsilon_2 2^{E_1 + E_2 - 2p}\alpha_1\alpha_2 - \varepsilon_3 2^{E_3 - p}\alpha_3 \tag{10}$$

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

$$X_1 \otimes X_2 = x_1 x_2 - \varepsilon_1 2^{E_1 - p}\alpha_1 x_2 - \varepsilon_2 2^{E_2 - p}\alpha_2 x_1 - \varepsilon_3 2^{E_3 - p}\alpha_3 \tag{11}$$

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

$$X_1 \oslash X_2 = \frac{x_1}{x_2} - \varepsilon_1 2^{E_1 - p}\frac{\alpha_1}{x_2} + \varepsilon_2 2^{E_2 - p}\alpha_2 \frac{x_1}{x_2^2} - \varepsilon_3 2^{E_3 - p}\alpha_3 \tag{12}$$

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 that result 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, then 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*.

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 modeled to the first order in 2^{-p} as:

$$R \approx r + \sum_{i=1}^{n} g_i(d) 2^{-p}\alpha_i \tag{13}$$

where r is the exact result, p is the number of bits in the mantissa, α_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. For instance, in Equation (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

$$C_{R,r} = \log_2 \left| \frac{R + r}{2(R - r)} \right| \tag{14}$$

$$C_{R,r} \approx -\log_2 \left| \frac{R - r}{r} \right| = p - \log_2 \left| \sum_{i=1}^{n} g_i(d) \frac{\alpha_i}{x} \right| \tag{15}$$

The last term in Equation (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 (13) is valid, the loss of accuracy in a computation is independent of the precision used.

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 the nearest,

$$(-10^{20} \oplus 10^{20}) \oplus 1 = 1 \tag{16}$$

but

$$-10^{20} \oplus (10^{20} \oplus 1) = 0 \tag{17}$$

Equation (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 the 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 obtains 41152264 and 41152268, respectively.

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, otherwise 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 that 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 the 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.

Impact on Iterative Methods. The result of an iterative method is defined as the limit L of a first-order recurrent sequence:

$$L = \operatorname*{Lim}_{n \to \infty} U_n \quad \text{with} \quad U_{n+1} = \mathcal{F}(U_n) \quad \mathbb{R}^m \xrightarrow{\mathcal{F}} \mathbb{R}^m \tag{18}$$

Because of rounding error propagation, the same problems as in a direct method may occur. But another difficulty is caused by 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:

$$\|U_n - U_{n-1}\| \leq \varepsilon \tag{19}$$

or the relative error:

$$\|U_n - U_{n-1}\| \leq \varepsilon \, \|U_{n-1}\| \tag{20}$$

It may be difficult to choose a suitable value for ε. If ε is too high, then computations stop too early and the result is very approximative. If ε 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 ε on the quality of the result is shown in the numerical experiment described below.

Newton's method is used to compute a root of

$$f(x) = x^4 - 1002x^3 + 252001x^2 - 501000x + 250000 \tag{21}$$

The following sequence is computed:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad \text{with} \quad x_0 = 1100 \tag{22}$$

The exact limit is $L = 500$, which is 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 ε the last value of n and the error $|x_n - L|$ computed using IEEE double precision arithmetic with rounding to the nearest.

It is noticeable that the optimal order of magnitude for ε 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 $\varepsilon = 10^{-11}$.

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, then the truncation error $E_t(h)$ also decreases, but the rounding error $E_r(h)$ usually increases, as shown in Fig. 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.

The numerical experiment described below (4) shows the impact of the step h on the quality of the approximation. The second derivative at $x = 1$ of the following function

$$f(x) = \frac{4970x - 4923}{4970x^2 - 9799x + 4830} \tag{23}$$

Table 2. Number of Iterations and Error Obtained Using Newton's Method in Double Precision

ε	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

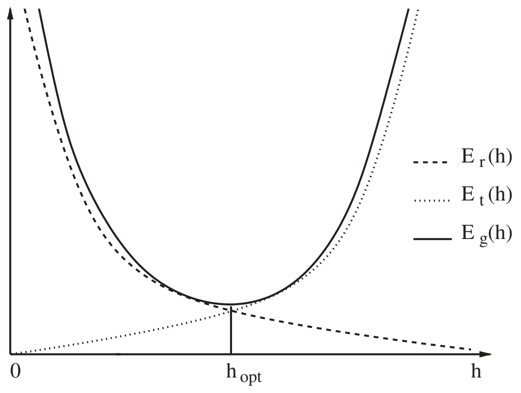

Figure 1. Evolution of the rounding error $E_r(h)$, the truncation error $E_t(h)$ and the global error $E_g(h)$ with respect to the step h.

is approximated by

$$L(h) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} \tag{24}$$

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 the nearest.

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

METHODS FOR ROUNDING ERROR ANALYSIS

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

Forward/Backward Analysis

This subsection is heavily inspired from Refs. 5 and 6. Other good references are Refs. 7–9.

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

$$E_a(X) = |x - X| \tag{25}$$

Table 3. Second Order Approximation of $f''(1) = 94$ Computed in Double Precision

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

and its *relative error*

$$E_r(X) = \frac{|x - X|}{|x|} \tag{26}$$

(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.

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, then 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.

Conditioning. Given a well-posed problem (P), one wants now to know how to measure the difficulty of solving this problem. This measurement 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. Because 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 by $\mathcal{D}$ and $\mathcal{R}$, respectively 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 Δy of the data y that satisfies $\|\Delta y\|_{\mathcal{D}} \le \varepsilon$, 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 nonzero. The *condition number* of the problem (P) in the data y is defined by

$$\operatorname{cond}(P, y) := \lim_{\varepsilon \to 0} \sup_{\Delta y \in \mathcal{P}(\varepsilon), \Delta y \neq 0} \left\{ \frac{\|\Delta x\|_{\mathcal{R}}}{\|\Delta y\|_{\mathcal{D}}} \right\} \tag{27}$$

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 $\|\Delta x\|_{\mathcal{R}} = |\Delta x|/|x|$. Because

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

one has[1]

$$\frac{\|\Delta x\|_{\mathcal{R}}}{\|\Delta y\|_{\mathcal{D}}} \leq \frac{\sum_{i=1}^{n} |y_i|}{|\sum_{i=1}^{n} y_i|} \tag{28}$$

This bound is reached for the perturbation Δy 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,

$$\text{cond}\left(\sum_{i=1}^{n} y_i\right) = \frac{\sum_{i=1}^{n} |y_i|}{|\sum_{i=1}^{n} y_i|} \tag{29}$$

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 a later section 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.

Stability of an Algorithm. Problems are generally solved using an *algorithm*, which 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 $\hat{G}$. Therefore, the algorithm does not compute $x = G(y)$ but $\hat{x} = \hat{G}(y)$.

The *forward analysis* tries to study the execution of the algorithm $\hat{G}$ 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 $\hat{x}$. This difference between the exact solution x and the computed solution $\hat{x}$ is called the *forward error*.

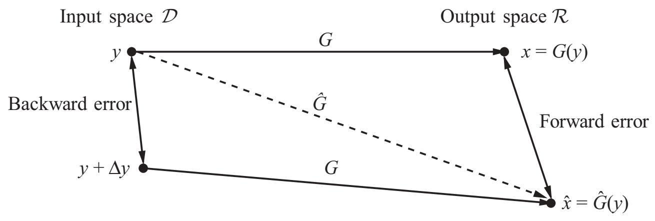

Figure 2. Forward and backward error for the computation of $x = G(y)$.

It is easy to recognize 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 Δy such that $\hat{x} = G(y + \Delta y)$. Δy is said to be the *backward error* associated with $\hat{x}$. A backward error measures the distance between the problem that is solved and the initial problem. As $\hat{x}$ and G are known, it is often possible to obtain a good upper bound for Δy (generally, it is easier than for the forward error). Figure 2 sums up the principle of the forward and backward analysis.

Sometimes, it is not possible to have $\hat{x} = G(y + \Delta y)$ for some Δy but it is often possible to get Δx and Δy such that $\hat{x} + \Delta x = G(y + \Delta y)$. 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 $\hat{x} = \hat{G}(y)$ is the scalar $\eta(\hat{x})$ defined by, when it exists,

$$\eta(\hat{x}) = \min_{\Delta y \in \mathcal{D}} \{\|\Delta y\|_{\mathcal{D}} : \hat{x} = G(y + \Delta y)\} \tag{30}$$

If it does not exist, then $\eta(\hat{x})$ is set to $+\infty$. An algorithm is said to be *backward-stable* for the problem (P) if the computed solution $\hat{x}$ has a "small" backward error $\eta(\hat{x})$. 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:

$$\hat{z} = z(1 + \delta) = (a + b)(1 + \delta) \text{ with } |\delta| \leq u \tag{31}$$

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 Equation (31), one can write

$$s_i = (s_{i-1} + y_i)(1 + \delta_i) \text{ with } |\delta_i| \leq u \tag{32}$$

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

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

$$\texttt{res} = y_1(1+\theta_{n-1}) + y_2(1+\theta_{n-1}) + y_3(1+\theta_{n-2}) + \cdots + y_{n-1}(1+\theta_2) + y_n(1+\theta_1) \quad (33)$$

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

$$\eta(\hat{x}) = |\theta_{n-1}| \lesssim nu \quad (34)$$

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

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*:

$$\text{forward error} \lesssim \text{condition number} \times \text{backward error} \quad (35)$$

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

$$\text{forward error} \lesssim \text{condition number} \times u \quad (36)$$

In general, the condition number is hard to compute (as hard as the problem itself). As a consequence, some estimators 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 were 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 $\hat{s}$ the computed sum of the vector y_i, $1 \le i \le n$ and

$$s = \sum_{i=1}^{n} y_i$$

the real sum, then Equation (33) implies

$$\frac{|\hat{s}-s|}{|s|} \le \gamma_{n-1}\,\mathrm{cond}\left(\sum_{i=1}^{n} y_i\right) \quad (37)$$

with γ_n defined by

$$\gamma_n := \frac{nu}{1-nu} \text{ for } n \in \mathbb{N} \quad (38)$$

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

The LAPACK Library. The LAPACK library (10) 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 established in Equation (35).

In fact, the major part of the algorithms implemented in LAPACK are backward stable, which means that the rule of thumb [Equation (36)] 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.

Interval Arithmetic

Interval arithmetic (11, 12) 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.

$$X + Y = [\underline{x} + \underline{y},\ \overline{x} + \overline{y}] \quad (39)$$

$$X - Y = [\underline{x} - \overline{y},\ \overline{x} - \underline{y}] \quad (40)$$

$$X \times Y = [\min(\underline{x}\times\underline{y},\ \underline{x}\times\overline{y},\ \overline{x}\times\underline{y},\ \overline{x}\times\overline{y}) \\ \max(\underline{x}\times\underline{y},\ \underline{x}\times\overline{y},\ \overline{x}\times\underline{y},\ \overline{x}\times\overline{y})] \quad (41)$$

$$X^2 = [\min(\underline{x}^2, \overline{x}^2), \max(\underline{x}^2, \overline{x}^2)] \text{ if } 0 \notin [\underline{x}, \overline{x}] \\ [0, \max(\underline{x}^2, \overline{x}^2)] \text{ otherwise} \quad (42)$$

$$1/Y = [\min(1/\underline{y}, 1/\overline{y}), \max(1/\underline{y}, 1/\overline{y})] \text{ if } 0 \notin [\underline{y}, \overline{y}] \quad (43)$$

$$X/Y = [\underline{x}, \overline{x}] \times (1/[\underline{y}, \overline{y}]) \text{ if } 0 \notin [\underline{y}, \overline{y}] \quad (44)$$

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

Table 4. Determinant of the Hilbert Matrix H of Dimension 8

	det(H)	#exact digits
IEEE double precision	2.73705030017821E-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

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:

$$\forall x \in X,\ f(x) \in f(X) \tag{45}$$

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 that result from other mathematically equivalent formulas.

Interval arithmetic enables one to control rounding errors automatically. On a computer, a real value that 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

$$X + Y = [\nabla(\underline{x} + \underline{y}), \Delta(\overline{x} + \overline{y})] \supseteq \{x + y | x \in X, y \in Y\} \tag{46}$$

$$X - Y = [\nabla(\underline{x} - \overline{y}), \Delta(\overline{x} - \underline{y})] \supseteq \{x - y | x \in X, y \in Y\} \tag{47}$$

where ∇ (respectively Δ) 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] and a Matlab toolbox, INTLAB,[3] are freely available.

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 caused by the loss of variable dependency. In interval arithmetic, several occurrences of the same variable are considered as different variables. For instance, let $X = [1,2]$,

$$\forall x \in X,\ x - x = 0 \tag{48}$$

but

$$X - X = [-1, 1] \tag{49}$$

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

$$f(x,y) = \frac{\sqrt{2}}{2}(x + y, y - x) \tag{50}$$

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 classic numerical algorithms can lead to overpessimistic 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

$$H_{ij} = \frac{1}{i + j - 1} \quad \text{for} \quad i = 1, \ldots, 8 \quad \text{and} \quad j = 1, \ldots 8 \tag{51}$$

computed:

- using the Gaussian elimination in IEEE double precision arithmetic with rounding to the 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 Ref. 8, p. 214.

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

The exact value of the determinant is

$$\det(\mathrm{H}) = \prod_{\mathrm{k}=0}^{7} \frac{(\mathrm{k}!)^3}{(8 + \mathrm{k})!} \tag{52}$$

Its 15 first exact significant digits are:

$$\det(\mathrm{H}) = 2.73705011379151\mathrm{E} - 33 \tag{53}$$

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

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

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

error with naive implementations of classic numerical algorithms in interval arithmetic. The algorithm for the inclusion of a determinant that is specific to interval arithmetic leads to a much thinner interval. Such interval algorithms exist in most areas of numerical analysis. Interval analysis can be used not only for reliable numerical simulations but also for computer assisted proofs (cf., for example, Ref. 8).

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 of elementary rounding errors. In the general case, it is impossible to describe each elementary error carefully 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 loses the exact description of the phenomena, but one may hope to get some global information like order of magnitude, frequency, and so on. It is exactly what is hoped for when using a probabilistic model of rounding errors.

For the mathematical model, remember the formula at the first order [Equation (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 toward $-\infty$ 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, then one obtains for N different runs, N different results on which a statistical test may be applied. This strategy 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 that is different from one result to another.

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 toward $-\infty$ 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, $i = 1, \ldots, N$,
- estimating the number of exact decimal significant digits of $\overline{R}$ with

$$C_{\overline{R}} = \log_{10}\left(\frac{\sqrt{N}|\overline{R}|}{\sigma\tau_{\beta}}\right) \tag{54}$$

where

$$\overline{R} = \frac{1}{N}\sum_{i=1}^{N} R_i \quad \text{and} \quad \sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}(R_i - \overline{R})^2 \tag{55}$$

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

From Equation (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 Refs. 13 and 14. The approximation at the first order in Equation (13) is essential for the validation of the CESTAC method. It has been shown that this approximation may be wrong only if multiplications or divisions involve nonsignificant values. A nonsignificant value is a computed result for which all the significant digits are affected by rounding errors. Therefore, one needs a dynamical control of multiplication and division during the execution of the code. This step leads to the synchronous implementation of the method (i.e., to the parallel computation of the N results R_i.) In this approach, a classic 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 ω 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 nonsignificant. 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 Discrete Stochastic Arithmetic (DSA). 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. DSA is implemented in the Control of Accuracy and Debugging for Numerical Applications (CADNA) library.[4] The CADNA library allows, during the execution of any code:

- the estimation of the error caused by 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.

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 that use multiprecision arithmetic.

Compensated Methods

Throughout this subsection, one assumes that the floating-point arithmetic adhers to IEEE 754 floating-point standard in rounding to the nearest. One also assume that no overflow nor underflow occurs. The material presented in this section heavily relies on Ref. (15).

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 $+, -, \times, \sqrt{}$ the approximation error of a floating-point operation is still a floating-point number:

$$\begin{array}{lll} x = a \oplus b & \Rightarrow \quad a + b = x + y & \text{with}\, y \in \mathbb{F}, \\ x = a \ominus b & \Rightarrow \quad a - b = x + y & \text{with}\, y \in \mathbb{F}, \\ x = a \otimes b & \Rightarrow \quad a \times b = x + y & \text{with}\, y \in \mathbb{F}, \\ x = a \oslash b & \Rightarrow \quad a = x \times b + y & \text{with}\, y \in \mathbb{F}, \\ x = \circled{\surd}\ (a) & \Rightarrow \quad a = x^2 + y & \text{with}\, y \in \mathbb{F} \end{array} \tag{56}$$

These example 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 and y is the rounding term. Fortunately, the quantities x and y in Equation (56) 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. (16). 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 (17). 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 three operations costs more than six operations. As a consequence, `TwoSum` is generally more efficient than `FastTwoSum` plus a branching.

Algorithm 4.2. (17). 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 (17) splits a floating-point number $a \in \mathbb{F}$ into two parts x and y such that

$$a = x + y \quad \text{with}\, |y| \leq |x| \tag{57}$$

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

Algorithm 4.3. (17) Error-free split of a floating-point number into two parts

```
function [$x$, $y$] = Split($a$)
  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

$$a \times b = x + y \quad \text{with } x = a \otimes b \tag{58}$$

Algorithm 4.4. (17). 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($b$)
  $y = a_2 \otimes b_2 \ominus (((x \ominus a_1 \otimes b_1) \ominus a2 \otimes b_1) \ominus a_1 \otimes b_2)$
```

The performance of the algorithms is interpreted in terms of floating-point operations (flops). The following

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

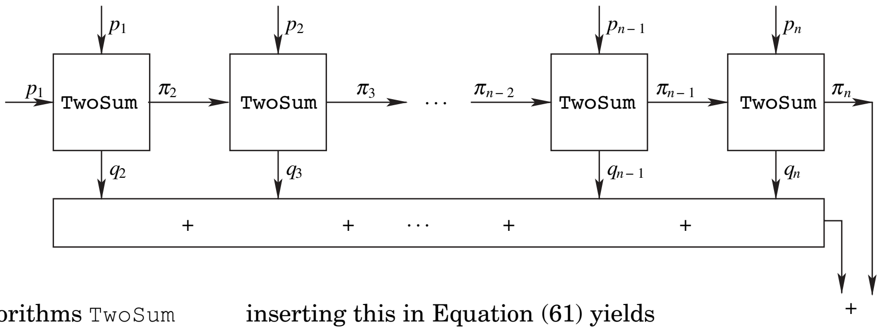

Figure 3. Compensated summation algorithm.

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,

$$a + b = x + y, \quad x = a \oplus b, \quad |y| \le u|x|, \quad |y| \le u|a + b|. \tag{59}$$

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,

$$a \times b = x + y, \quad x = a \otimes b, \quad |y| \le u|x|, \quad |y| \le u|a \times b|. \tag{60}$$

The algorithm TwoProduct requires 17 flops.

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 Fig. 3 and Algorithm 4.5.

Algorithm 4.5. Compensated summation algorithm

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

The following proposition gives a bound on the accuracy of the result. The notation γ_n defined by Equation (38) will be used. When using $\gamma_n, nu \le 1$ is implicitly assumed.

Proposition 4.2. (15). Suppose Algorithm CompSum is applied to floating-point number $p_i \in \mathbb{F}, 1 \le i \le n$. Let $s := \sum p_i, S := \sum |p_i|$ and nu < 1. Then, one has

$$|\mathrm{res} - s| \le u|s| + \gamma_{n-1}^2 S \tag{61}$$

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

One can interpret Equation (61) in terms of the condition number for the summation (29). Because

$$\mathrm{cond}\Big(\sum p_i\Big) = \frac{\sum |p_i|}{|\sum p_i|} = \frac{S}{|s|} \tag{62}$$

inserting this in Equation (61) yields

$$\frac{|\mathrm{res} - s|}{|s|} \le u + \gamma_{n-1}^2 \mathrm{cond}\Big(\sum p_i\Big) \tag{63}$$

Basically, the bound for the relative error of the result is essentially $(nu)^2$ times the condition number plus the rounding u because of 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 varied from 10^4 to 10^{40}.

Figure 4 shows the relative accuracy $|\mathrm{res} - s|/|s|$ of the computed value by the two algorithms 3.1 and 4.5. The *a priori* error estimations Equations (37) and (63) are also plotted.

As one can see in Fig. 4, the compensated summation algorithm exhibits the expected behavior, that is to say, the compensated rule of thumb Equation (63). 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}.

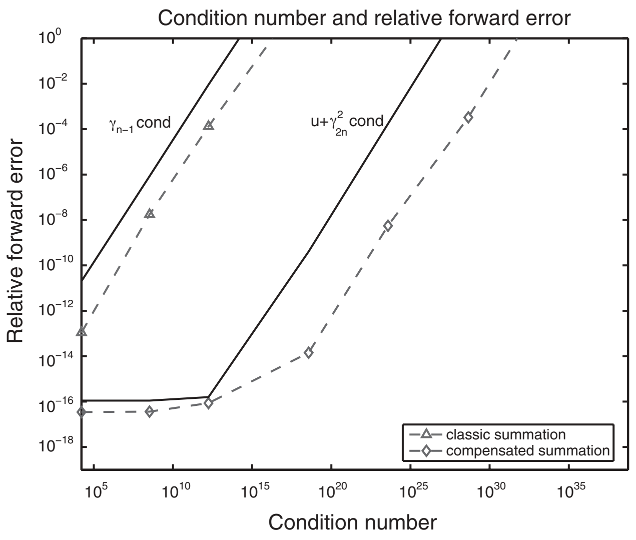

Figure 4. Compensated summation algorithm.

Multiple Precision Arithmetic

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

- Arbitrary precision libraries using a *multiple-digit* format in which a number is expressed as a sequence of digits coupled with a single exponent. Examples of this format are Bailey's MPFUN/ARPREC,[5] Brent's MP,[6] or MPFR.[7]
- Arbitrary precision libraries using a *multiple-component* format where a number is expressed as unevaluated sums of ordinary floating-point words. Examples using this format are Priest's[8] and Shewchuk's[9] libraries. Such a format is also introduced in Ref. 18.
- Extended fixed-precision libraries using the *multiple-component* format but with a limited number of components. Examples of this format are Bailey's *double-double*[5] (double-double numbers are represented as an unevaluated sum of a leading double and a trailing double) and *quad-double*.[5]

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, and so on.

Algorithm 4.6. 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 4.7. 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($t_h, (t_l \oplus s_l)$)
```

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

[6]http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub043.html.

[7]http://www.mpfr.org/.

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

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

Algorithm 4.8. 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 4.6 to 4.8 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 $|c_l| \leq u|c_h|$.

Several implementations can be used 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 (19) 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 [Equation (63)] (see Ref. 15), so it is an example for a double-double result.

Let us now briefly describe the MPFR library. This library is written in C language 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.

Unlike compensated methods that need to modify the algorithms, multiprecision libraries are convenient ways to increase the precision without too many efforts.

ACKNOWLEDGMENT

The authors sincerely wish to thank the reviewers for their careful reading and their constructive comments.

BIBLIOGRAPHY

1. report of the General Accounting office, GAO/IMTEC-92-26.
2. D. Goldberg, What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surve.*, **23**(1): 5–48, 1991.

3. IEEE Computer Society, *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. Reprinted in SIGPLAN Notices, **22**(2): 9–25, 1987.
4. S. M. Rump, How reliable are results of computers? *Jahrbuch Überblicke Mathematik*, pp. 163–168, 1983.
5. N. J. Higham, *Accuracy and stability of numerical algorithms*, Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 2nd ed. 2002.
6. P. Langlois, Analyse d'erreur en precision finie. In A. Barraud (ed.), *Outils d'Analyse Numérique pour l'Automatique*, Traité IC2, Cachan, France: Hermes Science, 2002, pp. 19–52.
7. F. Chaitin-Chatelin and V. Frayssé, *Lectures on Finite Precision Computations*. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 1996.
8. S. M. Rump, Computer-assisted proofs and self-validating methods. In B. Einarsson (ed.), *Accuracy and Reliability in Scientific Computing*, Software-Environments-Tools, Philadelphia, PA: SIAM, 2005, pp. 195–240.
9. J. H. Wilkinson, Rounding errors in algebraic processes. (32), 1963. Also published by Englewood Cliffs, NJ: Prentice-Hall, and New York: Dover, 1994.
10. 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*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.
11. G. Alefeld and J. Herzberger, *Introduction to Interval Analysis*. New York: Academic Press, 1983.
12. U. W. Kulisch, *Advanced Arithmetic for the Digital Computer*. Wien: Springer-Verlag, 2002.
13. J.-M. Chesneaux. *L'Arithmétique Stochastique et le Logiciel CADNA*. Paris: Habilitation à diriger des recherches, Université Pierre et Marie Curie, 1995.
14. J. Vignes, A stochastic arithmetic for reliable scientific computation. *Math. Comput. Simulation*, **35**: 233–261, 1993.
15. T. Ogita, S. M. Rump, and S. Oishi, Accurate sum and dot product. *SIAM J. Sci. Comput.*, **26**(6): 1955–1988, 2005.
16. D. E. Knuth, *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, 3rd ed. Reading, MA: Addison-Wesley, 1998.
17. T. J. Dekker, A floating-point technique for extending the available precision. *Numer. Math.*, **18**: 224–242, 1971.
18. 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, Germany: Hamburg University of Technology, 2007.
19. 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.

JEAN-MARIE CHESNEAUX
STEF GRAILLAT
FABIENNE JÉZÉQUEL
Laboratoire d'Informatique de Paris, France