An Overview of Semantics for the Validation of Numerical Programs

Matthieu Martel

CEA - Recherche Technologique
LIST-DTSI-SOL
CEA F91191 Gif-Sur-Yvette Cedex, France
e-mail : mmartel@cea.fr

Abstract. In this article, we introduce a simple formal semantics for floating-point numbers with errors which is expressive enough to be formally compared to the other methods. Next, we define formal semantics for interval, stochastic, automatic differentiation and error series methods. This enables us to formally compare the properties calculated in each semantics to our reference, simple semantics. Most of these methods having been developed to verify numerical intensive codes, we also discuss their adequacy to the formal validation of softwares and to static analysis. Finally, this study is completed by experimental results.

1 Introduction

Interval computations, stochastic arithmetics, automatic differentiation, etc.: much work is currently done to estimate and to improve the numerical accuracy of programs. Beside the verification of numerical intensive codes, which is the historical applicative domain of these methods, a new problematic is growing that concerns the formal validation of the accuracy of numerical calculations in critical embedded systems.

Despite the large amount of work in this area, few comparative studies have been carried out. This is partly due to the fact that the numerical properties calculated by different methods are difficult to relate. For example, how to compare results coming from interval arithmetics to the ones obtained by automatic differentiation?

This article attempts to clarify the links between the most commonly used methods among the above-mentioned ones. First, we introduce a simple formal semantics for floating-point numbers with errors which is expressive enough to be formally compared to the other methods. This semantics is a special instance of a familly of semantics introduced recently [22]. Next, we define formal semantics for interval, stochastic, automatic differentiation and error series methods which are usually expressed in other, less semantical, settings. This enables us to compare the properties calculated by each semantics to our reference semantics and to oversee how different methods could be coupled to obtain more accurate results.

Most of these methods having been developed to verify numerical intensive codes, we discuss their adequacy to the formal validation of critical systems.

From our point of view, a method is well suited for the validation of embedded applications if it enables the user to detect errors in an application that uses standard, non-instrumented, floating-point numbers, once embedded. In addition, we discuss the adequacy of the different semantics to static analysis. We complete our study by conducting some experiments. The methods described in this article are applied to simple examples, to show their ability and limits to detect numerical errors in C codes.

We limit our study to the semantics dealing with numerical precision. This excludes other interesting related works that also contribute to the validation of the numerical accuracy of softwares, like formal proof techniques of numerical properties over the floating-point numbers (e.g. [5, 10, 19]), or constraints solvers over the floating-point numbers which are used for structural test case generation [24]. It also excludes alternative arithmetics enabling to improve the accuracy of the float operations like multiple precision arithmetics [18, 29] or exact arithmetics [28]. These alternative arithmetics are more accurate than the standard floating-point numbers but they do not provide information on the precision of the results.

This article is organized as follows. Section 2 briefly presents some aspects of the IEEE 754 Standard. In Section 3, we introduce a simple semantics attaching to each floating-point number an error term measuring the distance to the exact real number which has been approximated. Next, this semantics is compared to other semantics, based on interval arithmetics (Section 4), stochastic arithmetics (Section 5), automatic differentiation (Section 6) and error series (Section 7). In Section 8, we discuss the adequacy of each method to static analysis and finally, in Section 9, we present experimental results illustrating how the techniques described in this article work on simple examples. Section 10 concludes.

2 Floating-point numbers

The IEEE 754 Standard specifies the representation of floating-point numbers as well as the behavior of the elementary operations [2, 11]. It is now implemented in almost all modern processors and, consequently, it provides a precise semantics, used as a basis in this article, for the basic operations occurring in high-level programming languages. First of all, a floating-point number x in base β is defined by

$$x = s \cdot (d_0.d_1 \ldots d_{p-1}) \cdot \beta^e = s \cdot m \cdot \beta^{e-p+1} \tag{1}$$

where $s \in \{-1, 1\}$ is the sign, $m = d_0 d_1 \ldots d_{p-1}$ is the mantissa with digits $0 \le d_i < \beta$[1], $0 \le i \le p-1$, p is the precision and e is the exponent, $e_{min} \le e \le e_{max}$. The IEEE Standard 754 specifies a few values for p, e_{min} and e_{max}. For example, simple precision numbers are defined by $\beta = 2$, $p = 23$, $e_{min} = -126$ and $e_{max} = +127$. the standard also defines special values like NAN (not a number) or $\pm\infty$. In this article, the notation $\mathbb{F}$ indifferently refers to the set

[1] $d_0 \neq 0$ but for denormalized numbers.

of simple or double precision numbers, since our assumptions conform to both types. $\mathbb{R}$ denotes the set of real numbers.

The standard defines four rounding modes for elementary operations between floating-point numbers. These modes are towards $-\infty$, towards $+\infty$, towards zero and to the nearest. We write them $\circ_{-\infty}$, $\circ_{+\infty}$, $\circ_0$ and $\circ_{\sim}$ respectively. Let $\uparrow_\circ : \mathbb{R} \to \mathbb{F}$ be the function which returns the roundoff of a real number following the rounding mode $\circ \in \{\circ_{-\infty}, \circ_{+\infty}, \circ_0, \circ_{\sim}\}$. $\uparrow_\circ$ is fully specified by the norm. The standard specifies the behavior of the elementary operations $\diamond \in \{+,\ -,\ \times,\ \div\}$ between floating-point numbers by

$$f_1 \diamond_{\mathbb{F},\circ} f_2 \ = \uparrow_\circ (f_1 \diamond_{\mathbb{R}} f_2) \tag{2}$$

In this article, we also use the function $\downarrow_\circ : \mathbb{R} \to \mathbb{R}$ which returns the roundoff error. We have $\downarrow_\circ (r) = r - \uparrow_\circ (r)$.

Many reasons may lead to an important loss of accuracy in a float computation. For example, a catastrophic cancellation arises when subtracting close approximate numbers x and y [11]. An absorption arises when adding two number of different magnitude $x \ll y$; in this case $x +_{\mathbb{F}} y = y$ with $x \neq 0$.

The error due to the roundoff of an initial datum or resulting from the roundoff of the result of an operation is called a first order error. When errors are multiplied together, we obtain higher order errors. For example $(x+\epsilon_x) \times (y+\epsilon_y) = xy + x\epsilon_y + y\epsilon_x + \epsilon_x\epsilon_y$. Here, $x\epsilon_y + y\epsilon_x$ is the new first order error and $\epsilon_x\epsilon_y$ is a second order error.

The errors arising during a float computation can be estimated in different ways. Let $g_{\mathbb{R}} \ : \ \mathbb{R} \to \mathbb{R}$ be a function of the reals and $g_{\mathbb{F}} \ : \ \mathbb{F} \to \mathbb{F}$ its implementation in the floating-point numbers. The forward error estimates, for a given input x the distance $d(g_{\mathbb{R}}(x), g_{\mathbb{F}}(x))$. The backward error $B(x)$ determines whether the approximated solution $g_{\mathbb{F}}(x)$ is the exact solution to a problem close to the original one [8]: $B(x) = \inf \left\{ d(x,y) \ : \ y = g_{\mathbb{R}}^{-1}(g_{\mathbb{F}}(x)) \right\}$

Most of the existing automatic methods (and all the methods used in this article) compute forward errors.

3 Global error

In this section, we introduce the semantics $[\![.]\!]_{\mathbb{E}}$ which is used as a reference in the rest of this article. $[\![.]\!]_{\mathbb{E}}$ computes the floating-point number resulting from a calculation on a IEEE 754 compliant computer as well as the error arising during the execution. In other words, this semantics calculates the forward error, as defined in Section 2, between the exact result of a problem in the reals and the approximated solution returned by a program. To calculate the exact errors, $[\![.]\!]_{\mathbb{E}}$ uses real numbers and, consequently, it remains a theoretical tool. This semantics corresponds to the semantics $\mathcal{S}^{\mathcal{L}'}$ introduced in [22].

Formally, in $[\![.]\!]_{\mathbb{E}}$, a value v is denoted by a two-dimensional vector $v = f\boldsymbol{\varepsilon}_f + e\boldsymbol{\varepsilon}_e$. $f \in \mathbb{F}$ denotes the float used by the machine and $e \in \mathbb{R}$ denotes the exact error attached to f. $\boldsymbol{\varepsilon}_f$ and $\boldsymbol{\varepsilon}_e$ are formal variables used to identify the float and error components of v. For example, in simple precision, using the

$$x_1 = f_1\varepsilon_f + e_1\varepsilon_e \quad \text{and} \quad x_2 = f_2\varepsilon_f + e_2\varepsilon_e \tag{3}$$

$$x_1 + x_2 = \uparrow_\circ (f_1 + f_2)\varepsilon_f + [e_1 + e_2 + \downarrow_\circ (f_1 + f_2)]\,\varepsilon_e \tag{4}$$

$$x_1 - x_2 = \uparrow_\circ (f_1 - f_2)\varepsilon_f + [e_1 - e_2 + \downarrow_\circ (f_1 - f_2)]\,\varepsilon_e \tag{5}$$

$$x_1 \times x_2 = \uparrow_\circ (f_1 \times f_2)\varepsilon_f + [e_1 f_2 + e_2 f_1 + e_1 e_2 + \downarrow_\circ (f_1 \times f_2)]\,\varepsilon_e \tag{6}$$

$$\frac{1}{x_1} = \uparrow_\circ \left(\frac{1}{f}\right)\varepsilon_f + \left[\downarrow_\circ \left(\frac{1}{f}\right) + \sum_{n\geq 1}(-1)^n \frac{e^n}{f^{n+1}}\right]\varepsilon_e \tag{7}$$

Fig. 1. *The semantics* $[\![.]\!]_{\mathbb{E}}$.

functions $\uparrow_\circ$ and $\downarrow_\circ$ introduced in Section 2, the real number $\frac{1}{3}$ is represented by the value $v = \uparrow_\circ (\frac{1}{3})\varepsilon_f + \downarrow_\circ (\frac{1}{3})\varepsilon_e = 0.333333\varepsilon_f + (\frac{1}{3} - 0.333333)\varepsilon_e$. The semantics interprets a constant d as follows:

$$[\![d]\!]_{\mathbb{E}} = \uparrow_\circ (d)\varepsilon_f + \downarrow_\circ (d)\varepsilon_e \tag{8}$$

The semantics of elementary operations is defined in Figure 1, the operands x_1 and x_2 being given in Equation (3). Equations (4–6) are immediate. For Equation (7), recall that $\frac{1}{1+x} = \sum_{n\geq 0}(-1)^n x^n$ for all x such that $-1 \leq x \leq 1$. We have:

$$\frac{1}{f+e} = \frac{1}{f} \times \frac{1}{1+\frac{e}{f}} = \frac{1}{f} \times \sum_{n\geq 0}(-1)^n \frac{e^n}{f^n}$$

The power series development is valid for $-1 \leq \frac{e}{f} \leq 1$ or, equivalently, while $|e| \leq |f|$, i.e. as long as the error is less than the float in absolute value. The semantics of the square root function is obtained like for division but the other elementary functions (e.g. the trigonometric ones) are more difficult to handle, due to the fact that the IEEE 754 Standard does not specify how they are rounded. $[\![.]\!]_{\mathbb{E}}$ calculates the floating-point numbers returned by a program and the exact difference between the float and real results, as outlined by the following proposition.

Proposition 1 *Let a be an arithmetic expression. Then if $[\![a]\!]_{\mathbb{E}} = f\varepsilon_f + e\varepsilon_e$ then $[\![a]\!]_{\mathbb{F}} = f$ and $[\![a]\!]_{\mathbb{R}} = f + e$.*

By relating $[\![.]\!]_{\mathbb{E}}$ to the semantics $[\![.]\!]_{\mathbb{R}}$ of real numbers and to the semantics $[\![.]\!]_{\mathbb{F}}$ of floating-point numbers, Proposition 1 provides a correctness criterion for $[\![.]\!]_{\mathbb{E}}$.

$[\![.]\!]_{\mathbb{E}}$ is well suited for the formal validation of critical systems because it exactly gives the floating-point numbers f used by the non-instrumented embedded code running on a IEEE 754 compliant computer as well as the error e arising when f is used instead of the exact value $f + e$. However, this semantics remains a theoretical tool since it uses real numbers and the function $\downarrow_\circ$ which cannot be exactly calculated by a computer in general. In the next sections, we use it as a reference in the study of other, approximate, semantics. We will compare the other methods in their ability to estimates the quantities f and e that define the values $f\varepsilon_f + e\varepsilon_e$ of $[\![.]\!]_{\mathbb{E}}$. The link between $[\![.]\!]_{\mathbb{E}}$ and the other methods is summed up by propositions 2, 3, 4 and 5.

4 Intervals

The classical semantics of intervals $[\![.]\!]_{\mathbb{I}}$ aims at bounding the real result of a calculation by a lower and an upper float value [27]. Obviously, the semantics of a constant d is

$$[\![d]\!]_{\mathbb{I}} = [\uparrow_{-\infty}(d), \uparrow_{+\infty}(d)] \tag{9}$$

Similarly, let $x_1 = [\underline{x_1}, \overline{x_1}]$ and $x_2 = [\underline{x_2}, \overline{x_2}]$ be two float intervals, let $\diamond$ be an elementary operation, and let $i = [\underline{i}, \overline{i}]$ be the interval with real bounds defined by $i = x_1 \diamond x_2$. $[\![x_1 \diamond x_2]\!]_{\mathbb{I}}$ is defined by:

$$[\![x_1 \diamond x_2]\!]_{\mathbb{I}} = [\uparrow_{-\infty}(\underline{i}), \uparrow_{+\infty}(\bar{i})] \tag{10}$$

Basically, an interval computation bounds a real number by two floating-point numbers, the maximal float smaller or equal to the real and the minimal float greater or equal the real. In other terms, using the formalism of Section 3, an interval computation approximates from below and from above the sum $f + e$ corresponding to the float f and to the exact error e calculated by $[\![.]\!]_{\mathbb{E}}$. This is summed up in the following proposition.

Proposition 2 *Let a be an arithmetic expression such that $[\![a]\!]_{\mathbb{E}} = f\varepsilon_f + e\varepsilon_e$ and $[\![a]\!]_{\mathbb{I}} = [\underline{x}, \overline{x}]$. Then we have $[\![f + e]\!]_{\mathbb{I}} \subseteq [\underline{x}, \overline{x}]$.*

Note that if the interval bounds are expressed with the same precision as in the original program, as in the semantics defined by equations (9) and (10), then the result $[\underline{x}, \overline{x}]$ output by the interval method bounds both the float result f and the real result $f + e$. Otherwise, if the interval bounds are expressed with a greater precision than in the non-instrumented code then the interval method bounds the real result $f + e$ but not necessary the float result f. In this latter case, the method does not enable one to predict how a program behaves when using standard floating-point numbers.

Because $[\![.]\!]_{\mathbb{I}}$ always adds the error terms to the floating-point numbers, an interval computation does not distinguish two different kinds of errors:

1. Sensivity errors, due to the fact that a small variation of the inputs may yield a large variation of the outputs, even with real numbers.
2. Numerical errors, due to the fact that a float calculation may diverge from a real calculation.

For example, numerical errors arise in the following program which uses simple precision floating-point numbers:

```
float x=1.0;
float y=1.0e-8;
for(int i=0;i<1e8;i++) {
  x=x-y;
}
```

The value 10^{-8} being subtracted 10^8 times to 1, the exact result in the reals is $x_{\mathbb{R}} = 0$. But 10^{-8} is less than the least significant digit of the float 1.0 and, consequently, an absorption occurs: $1.0 - 10^{-8} = 1.0$ in the floating-point numbers. So, at the end of the iteration $x_{\mathbb{F}} = 1.0$. The interval semantics defined by equations (9) and (10) returns an interval $x_{\mathbb{I}} \supseteq [0, 1]$, indicating that the exact result is between 0 and 1. In $[\![.]\!]_{\mathbb{E}}$, $x_{\mathbb{E}} = 1.0\boldsymbol{\varepsilon}_f - 1.0\boldsymbol{\varepsilon}_e$ which means that the float value of x is 1 and that the exact forward error on x is -1.

As illustrated by our example, $[\![.]\!]_{\mathbb{I}}$ provides less information than the semantics of global errors because no distinction is made between the float and error terms. Nevertheless, when the interval resulting from a calculation is small, we may conclude that the error term also is small. In this case, an interval method can validate a calculation. In addition, intervals can be used to implement the theoretical semantics $[\![.]\!]_{\mathbb{E}}$, yielding a new semantics $[\![.]\!]_{\mathbb{EI}}$. A value $v = f\boldsymbol{\varepsilon}_f + [\underline{x}, \overline{x}]\boldsymbol{\varepsilon}_e$ of $[\![.]\!]_{\mathbb{EI}}$ is made of a float f and an interval of error $[\underline{x}, \overline{x}]$. The elementary operations are defined by the rules of Figure 1, in which the computations on error terms are carried out in $[\![.]\!]_{\mathbb{I}}$.

Examples of interval arithmetic libraries are Boost [6] and MPFI [30], the latter being based on the multiple precision library MPFR [18]. Implementations of multiple precision interval libraries are compared in [17].

5 Stochastic arithmetics

Stochastic arithmetics consists of running a few times the same program, the roundoff errors being, at each run, randomly propagated. The common digits of the results of all the executions are assumed exact [9, 31]. The stochastic arithmetic semantics $[\![.]\!]_{\mathbb{S}}$, introduces the random roundoff function $\uparrow_? : \mathbb{F} \to \mathbb{F}$ defined by:

$$\uparrow_? (d) = \begin{cases} \text{either } \uparrow_{-\infty} (d) \\ \text{or } \uparrow_{+\infty} (d) \end{cases} \quad \text{with probability } \frac{1}{2} \tag{11}$$

In stochastic arithmetics, the n executions of a program are usually carried out synchronously, to cope with control flow problems, like ensuring that all the executions take the same branches. So, a stochastic value is a n-tuple containing the n values assigned to a number. A constant d is interpreted by:

$$[\![d]\!]_{\mathbb{S}} = (\uparrow_? (d), \ldots, \uparrow_? (d)) \tag{12}$$

and, for any elementary operation $\diamond$, we have:

$$[\![x \diamond x']\!]_{\mathbb{S}} = (\uparrow_? (x_1 \diamond x'_1), \ldots, \uparrow_? (x_n \diamond x'_n)) \tag{13}$$

$[\![.]\!]_{\mathbb{S}}$ uses the fact that roundoff errors usually cancel each other. This enables the user to obtain less pessimistic results than, e.g., with interval arithmetics. More precisely, the mean $\overline{x}$ of the result $x = (x_1, \ldots, x_n)$ of the n runs approximates the real result $x_{\mathbb{R}}$ of the calculation. Let $C(x, x_{\mathbb{R}})$ denote the number of digits common to $\overline{x}$ and $x_{\mathbb{R}}$. Using Student's test, the method makes it possible to compute $C(x, x_{\mathbb{R}})$ with probability P [9].

Proposition 3 *Let a be an expression such that* $[\![a]\!]_{\mathbb{S}} = (x_1, \ldots, x_n)$ *and* $[\![a]\!]_{\mathbb{E}} = f\boldsymbol{\varepsilon}_f + e\boldsymbol{\varepsilon}_e$*. Then, with probability* P*,* $\overline{x}$ *and* $f+e$ *have* $C(\overline{x}, f+e)$ *common digits, where:* $C(\overline{x}, f+e) = \log_{10}\left(\frac{\sqrt{n}|\overline{x}|}{\sigma\tau_P}\right)$ $\qquad \sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$

For $n = 3$ and $P = 0.95$, $\tau_P = 4,303$. However, Proposition 3 is based on the hypothesis that the roundoff errors are uniform and independent, the latter meaning that the errors arising at each step of a calculation are not correlated with each other. This is not always the case, mainly for loops. For example, in the program of Section 4, the same roundoff error is made at each iteration. In addition, Proposition 3 assumes that higher order errors are negligible with respect to the first order errors. In Section 6, we introduce an example for which this assumption does not hold.

Because $[\![.]\!]_{\mathbb{S}}$ approximates, with probability P, the exact result of a program p in the reals, it does not enable the user to ensure that no precision loss arises in the non-instrumented execution of p which uses standard floating-point numbers. However, even if this method is mainly taylored to detect stability problems in the algorithms used in a program, it can assert the validity of a floating-point calculation when (1) $C(\overline{x}, x_{\mathbb{R}})$ is high and (2) $\overline{x}$ is close to the float result f.

An issue to improve a stochastic arithmetics for validation would be to define a new semantics $[\![.]\!]_{\mathbb{ES}}$ based on $[\![.]\!]_{\mathbb{E}}$. In this new semantics, the exact error terms of $[\![.]\!]_{\mathbb{E}}$ would be computed in $[\![.]\!]_{\mathbb{S}}$. A value $v = f\boldsymbol{\varepsilon}_f + e\boldsymbol{\varepsilon}_e$ of $[\![.]\!]_{\mathbb{ES}}$ would be made of a float f and an error e which would be a stochastic number of $[\![.]\!]_{\mathbb{S}}$. However, the hypotheses and the correctness proofs of $[\![.]\!]_{\mathbb{S}}$ must be revisited. The CADNA library implements stochastic arithmetics [7].

6 Automatic differentiation

In this section we introduce a simple semantics performing an automatic differentiation of programs. More elaborated techniques are described in the references mentioned later on. In automatic differentiation [3, 16], one considers that a program p calculates a function g of the data for which we are going to evaluate, at the same time as g, the numerical values of the derivatives. If $d_1, \ldots, d_n$ denote the data used in p, then the program calculates the final values $v_1, \ldots, v_m$ such that:

$$\begin{pmatrix} v_1 \\ \vdots \\ v_m \end{pmatrix} = \begin{pmatrix} g_1(d_1, \ldots, d_n) \\ \vdots \\ g_m(d_1, \ldots, d_n) \end{pmatrix} \tag{14}$$

$v_1, \ldots, v_m$ are the final results of the program, at the end of the execution. For each $1 \le i \le m$, v_i is a function g_i of the data $d_1, \ldots, d_n$. Automatic differentiation aims at numerically calculating, in addition to the terms v_i, the partial derivatives $\frac{\partial g_i}{\partial d_j}(d_1, \ldots, d_n)$ for all $1 \le j \le n$. By determining whether a slight modification of the initial value d_j implies a large modification of the result v_i, the partial derivative $\frac{\partial g_i}{\partial d_j}(d_1, \ldots, d_n)$ indicates the sensitivity of v_i to the variations of d_j. If $\frac{\partial g_i}{\partial d_j}(d_1, \ldots, d_n) \approx 0$, v_i is not much sensitive to a variation

$$v_1 = (f_1, \delta_1, \ldots, \delta_n) \quad \text{and} \quad v_2 = (f_2, \eta_1, \ldots, \eta_n) \tag{17}$$

$$[\![v_1 + v_2]\!]_{\mathbb{D}} = (f_1 + f_2, \delta_1 + \eta_1, \delta_2 + \eta_2, \ldots, \delta_n + \eta_n) \tag{18}$$

$$[\![v_1 - v_2]\!]_{\mathbb{D}} = (f_1 - f_2, \delta_1 - \eta_1, \delta_2 - \eta_2, \ldots, \delta_n - \eta_n) \tag{19}$$

$$[\![v_1 \times v_2]\!]_{\mathbb{D}} = (f_1 \times f_2, f_1\eta_1 + f_2\delta_1, f_1\eta_2 + f_2\delta_2, \ldots, f_1\eta_n + f_2\delta_n) \tag{20}$$

$$[\![\frac{v_1}{v_2}]\!]_{\mathbb{D}} = \left(\frac{f_1}{f_2}, \frac{f_2\delta_1 - f_1\eta_1}{f_2^2}, \frac{f_2\delta_2 - f_1\eta_2}{f_2^2}, \ldots, \frac{f_2\delta_n - f_1\eta_n}{f_2^2}\right) \tag{21}$$

Fig. 2. *The semantics $[\![.]\!]_{\mathbb{D}}$ for automatic differentiation.*

of d_j. If $\frac{\partial g_i}{\partial d_j}(d_1, \ldots, d_n) > 1$ then the error on d_j is magnified in v_i by a factor approximately equal to $\frac{\partial g_i}{\partial d_j}(d_1, \ldots, d_n)$.

The most intuitive way to calculate the derivatives is to achieve a linear approximation of order one by means of the well-known formula:

$$g'(x) \approx \frac{g(x + \Delta x) - g(x)}{\Delta x} \tag{15}$$

However this method yields imprecise results, due to the fact that Δx is usually small. Instead, automatic derivation techniques calculate the derivatives by composing elementary functions, according to the chain rule:

$$(f \circ g)'(x) = \big[f(g(x))\big]' = g'(x) \times f'(g(x)) \tag{16}$$

Each function g_i is viewed as a chain of elementary functions such as additions, products, trigonometric functions, etc. The derivatives $\frac{\partial g_i}{\partial d_j}(d_0, \ldots, d_n)$ are calculated by the rule of Equation (16).

The semantics $[\![.]\!]_{\mathbb{D}}$ of Figure 2 achieves a sensitivity analysis to the data of a program p, by automatic differentiation. For a constant d_i, we have

$$[\![d_i]\!]_{\mathbb{D}} \;=\; \big(d_i, \delta_1 = 0, \ldots, \delta_i = 1, \ldots, \delta_n = 0\big) \tag{22}$$

A numerical value v of $[\![.]\!]_{\mathbb{D}}$ is a $(n+1)$-tuple $(f, \delta_1, \ldots, \delta_n)$. Intuitively, at some stage of the execution of p, v is the result of an intermediate calculation. In other terms, for a certain function g, the program p has calculated $h_1(d_1, \ldots, d_n)$ such that $g((d_1, \ldots, d_n)) = h_2 \circ h_1(d_1, \ldots, d_n)$ for some function h_2 and the current value v of $[\![.]\!]_{\mathbb{D}}$ represents

$$v = (f, \delta_1, \ldots, \delta_n) = \left(h_1(d_1, \ldots, d_n), \frac{\partial h_1}{\partial d_1}(d_1, \ldots, d_n), \ldots, \frac{\partial h_1}{\partial d_n}(d_1, \ldots, d_n)\right) \tag{23}$$

Automatic differentiation can be viewed as a way to approximately calculate the error term $e\boldsymbol{\varepsilon}_e$ of the semantics $[\![.]\!]_{\mathbb{E}}$. Given a program that implements a function g, $[\![.]\!]_{\mathbb{E}}$ calculates:

$$[\![g(d_1, \ldots, d_n)]\!]_{\mathbb{E}} = x_r = f_r\boldsymbol{\varepsilon}_f + e_r\boldsymbol{\varepsilon}_e \tag{24}$$

In the simplest case $m = n = 1$, i.e. for a program p calculating a function g of a single datum d_1, we can estimate the error term e_r of Equation (24) from the numerical values $g(d_1)$ and $\frac{\partial g}{\partial d_1}(d_1)$ returned by $[\![g(d_1)]\!]_{\mathbb{D}}$. Let e_{d_1} denote the initial error on the argument d_1 of g. By linear approximation, we have

$$x_r \approx g(d_1)\boldsymbol{\varepsilon}_f + \left(e_{d_1} \times \frac{\partial g}{\partial d_1}(d_1)\right)\boldsymbol{\varepsilon}_e \tag{25}$$

In Equation (25), the exact error term e_r is approximated by $e_{d_1} \times \frac{\partial g}{\partial d_1}(d_1)$.

In the general case, i.e. if $m \geq 1$ and $n \geq 1$, we have the following property.

Proposition 4 *Let* $e_1, \ldots, e_n$ *be the initial errors attached to data* $d_1, \ldots, d_n$ *and let* $v_1, \ldots, v_m$ *be the results of a computation as defined in Equation (14). If in the semantics* $[\![.]\!]_{\mathbb{E}}$*, for all* $1 \leq i \leq m$, $v_i = [\![g_i(d_1, \ldots, d_n)]\!]_{\mathbb{E}} = x_i\boldsymbol{\varepsilon}_f + e_{r_i}\boldsymbol{\varepsilon}_e$ *then, in* $[\![.]\!]_{\mathbb{D}}$, $v_i = [\![g_i(d_1, \ldots, d_n)]\!]_{\mathbb{D}} = (y_i, \delta_{i,1}, \ldots, \delta_{i,n})$ *such that* $x_i = y_i$ *and such that the error term* e_{r_i} *on the final result* v_i *is linearly approximated by:*

$$e_{r_i} \approx \sum_{1 \leq j \leq n} e_j \times \delta_{i,j} \tag{26}$$

The main drawback of automatic differentiation stems from the linear approximation made in Equation (25), which may under-estimate the errors (or over-estimate them, though it is usually less critical for validation). For example, let us consider the function:

```
float g(float x,int n) {
  y=x;
  for (int i=0;i<n;i++) { y=y*x; };
  return y;
}
```

In this function, if the parameter $x = f\boldsymbol{\varepsilon}_f + e\boldsymbol{\varepsilon}_e$ is such that $f < 1$ and $f + e > 1$ then, in the floating-point numbers, $g(x) \to 0$ as $n \to \infty$, while in the reals, $g(x) = g(f + e) \to \infty$ as $n \to \infty$. In $[\![.]\!]_{\mathbb{D}}$, the value returned by $\mathsf{g}(x, n)$ is (f^{n+1}, nf^n), where the component nf^n gives the sensitivity of g to the parameter x. If $f < 1$ then $nf^n \to 0$ as $n \to \infty$ and the approximation of Equation (26) may become irrelevant. For instance, if $x = 0.95\boldsymbol{\varepsilon}_f + 0.1\boldsymbol{\varepsilon}_e$ then on one hand we have $[\![\mathsf{g}(x, n)]\!]_{\mathbb{E}} \to 0\boldsymbol{\varepsilon}_f + \infty\boldsymbol{\varepsilon}_e$ as $n \to \infty$ while, on the other hand $[\![\mathsf{g}(x, n)]\!]_{\mathbb{D}} \to (0, 0)$ as $n \to \infty$ In this example, Equation (26) leads to the erroneous conclusion that for $x = 0.95\boldsymbol{\varepsilon}_f + 0.1\boldsymbol{\varepsilon}_e$, $e_x \approx 0$.

In addition, automatic differentiation only takes care of the errors on the initial data, neglecting the errors introduced by the operations, during the calculation. For example, from Equation (4), the semantics $[\![.]\!]_{\mathbb{E}}$ of an addition is:

$$x_1 + x_2 = \uparrow_\circ (f_1 + f_2)\boldsymbol{\varepsilon}_f + [e_1 + e_2 + \downarrow_\circ (f_1 + f_2)]\,\boldsymbol{\varepsilon}_e$$

$[\![.]\!]_{\mathbb{D}}$ makes it possible to estimate the terms e_1 and e_2 but neglects the error introduced by the addition itself, namely $\downarrow_\circ (f_1 + f_2)$. Finally, the higher-order error terms also are neglected. For example, the term $e_1 e_2$ occuring in the result of the product of Equation (6) is ignored.

Bischof et al. have recently published an overview of the implementations of automatic differentiation libraries [3]. In certain cases, automatic differentiation can also be used to improve the precision of a calculation, by adding correcting terms to the floating-point numbers computed by the machine [20, 21]. In this case, roundoff errors introduced by the elementary operations are not neglected and one can guarantee the precision of the final result.

7 Error series

In this section, we introduce the semantics $[\![.]\!]_{\mathbb{W}}$ of error series [13, 22]. The semantics $[\![.]\!]_{\mathbb{E}}$ of Section 3 globally computes the difference between the float and the real result of a calculation. However, when this error term is large, no hint is given to the programmer concerning its source. $[\![.]\!]_{\mathbb{W}}$ is a generalization of $[\![.]\!]_{\mathbb{E}}$ in which the roundoff errors arising at any stage of a calculation are traced, in order to detect which of them mainly contribute to the global error.

$[\![.]\!]_{\mathbb{W}}$ assumes that the control points of the program are annotated by unique labels $\ell \in \mathcal{L}$. A value of $[\![.]\!]_{\mathbb{W}}$ is a series

$$r = f\boldsymbol{\varepsilon} + \sum_{\ell \in \mathcal{L}} \omega^{\ell} \boldsymbol{\varepsilon}_{\ell} \tag{27}$$

Error series generalize the values of $[\![.]\!]_{\mathbb{E}}$. In Equation (27), f is the float approximating the value of r. f is always attached to the formal variable $\boldsymbol{\varepsilon}$ whose index is the empty word. A term $\omega^{\ell}\boldsymbol{\varepsilon}_{\ell}$ denotes the contribution to the global error of the first-order error introduced by the operation labeled ℓ during the evaluation of r. $\omega^{\ell} \in \mathbb{R}$ is the scalar value of this error term and $\boldsymbol{\varepsilon}_{\ell}$ is a formal variable. A special label hi which corresponds to no particular control point is used to identify the higher order errors. This comes from previous work [22] which introduces more general semantics for error series. Error terms of order n correspond to words of length n and the empty word ν is related to the term for the floating-point number. The multiplication of terms of order m and n yields a new term of order $m + n$ denoted by a word of length $m + n$. In this article, hi identifies all the words of length greater that one and the product of formal variables is defined by Equation (28).

$$\boldsymbol{\varepsilon}_u \times \boldsymbol{\varepsilon}_v = \begin{cases} \boldsymbol{\varepsilon}_{uv} \text{ if length}(uv) \le 1 \\ \boldsymbol{\varepsilon}_{hi} \text{ otherwise} \end{cases} \tag{28}$$

The elementary operations are defined in Figure 3 for $r_1 = f_1\boldsymbol{\varepsilon}_1 + \sum_{\ell \in \mathcal{L}^+} \omega_1^{\ell}\boldsymbol{\varepsilon}_{\ell}$ and $r_2 = f_2\boldsymbol{\varepsilon}_2 + \sum_{\ell \in \mathcal{L}^+} \omega_2^{\ell}\boldsymbol{\varepsilon}_{\ell}$. $\mathcal{L}^+$ denotes the set $\mathcal{L}$ without the empty word. In addition, the symbols f and ω are used interchangeably to denote the coefficient of the variable $\boldsymbol{\varepsilon}$. The formal series $\sum_{\ell \in \mathcal{L}} \omega^{\ell}\boldsymbol{\varepsilon}_{\ell}$ related to the result of an operation $\diamond^{\ell_i}$ contains the combination of the errors on the operands plus a new error term $\downarrow_{\circ} (f_1 \diamond f_2)\boldsymbol{\varepsilon}_{\ell_i}$ corresponding to the error introduced by the operation

$$r_1 +^{\ell_i} r_2 \stackrel{\text{def}}{=} \uparrow_\circ (f_1 + f_2)\varepsilon + \sum_{\ell \in \mathcal{L}^+} (\omega_1^\ell + \omega_2^\ell)\varepsilon_\ell + \downarrow_\circ (f_1 + f_2)\varepsilon_{\ell_i} \tag{29}$$

$$r_1 -^{\ell_i} r_2 \stackrel{\text{def}}{=} \uparrow_\circ (f_1 - f_2)\varepsilon + \sum_{\ell \in \mathcal{L}^+} (\omega_1^\ell - \omega_2^\ell)\varepsilon_\ell + \downarrow_\circ (f_1 - f_2)\varepsilon_{\ell_i} \tag{30}$$

$$r_1 \times^{\ell_i} r_2 \stackrel{\text{def}}{=} \uparrow_\circ (f_1 f_2)\varepsilon + \sum_{\substack{\ell_1 \in \mathcal{L},\ \ell_\in \in \mathcal{L} \\ \ell_1 \cdot \ell_2 \neq \nu}} \omega_1^{\ell_1}\omega_2^{\ell_2}\varepsilon_{\ell_1 \cdot \ell_2} + \downarrow_\circ (f_1 f_2)\varepsilon_{\ell_i} \tag{31}$$

$$(r_1)^{-1^{\ell_i}} \stackrel{\text{def}}{=} \uparrow_\circ (f_1^{-1})\varepsilon - \frac{1}{f_1}\sum_{\ell \in \mathcal{L}} \frac{\omega^\ell}{f_1}\varepsilon_\ell + \frac{1}{f_1}\sum_{n \geq 2}(-1)^n \left(\sum_{\ell \in \mathcal{L}} \frac{\omega^\ell}{f_1}\right)^n \varepsilon_{hi} + \downarrow_\circ (f_1^{-1})\varepsilon_{\ell_i} \tag{32}$$

$$r_1 \div^{\ell_i} r_2 \stackrel{\text{def}}{=} r_1 \times^{\ell_i} (r_2)^{-1^{\ell_i}} \tag{33}$$

Fig. 3. *Elementary operations for floating-point numbers with errors.*

$\diamond$ occurring at point ℓ_i. The rules for addition and subtraction are natural. The elementary errors are added or subtracted componentwise in the formal series and the new error due to point ℓ_i corresponds to the roundoff of the result.

Multiplication requires more care because it introduces higher-order errors due to the multiplication of the first-order errors. For instance, let us consider the multiplication at point ℓ_3 of two initial data $r_1^{\ell_1} = (f_1\varepsilon + \omega_1^{\ell_1}\varepsilon_{\ell_1})$ and $r_2^{\ell_2} = (f_2\varepsilon + \omega_2^{\ell_2}\varepsilon_{\ell_2})$:

$$r_1^{\ell_1} \times^{\ell_3} r_2^{\ell_2} = \uparrow_\circ (f_1 f_2)\varepsilon + f_2\omega_1^{\ell_1}\varepsilon_{\ell_1} + f_1\omega_2^{\ell_2}\varepsilon_{\ell_2} + \omega_1^{\ell_1}\omega_2^{\ell_2}\varepsilon_{hi} + \downarrow_\circ (f_1 f_2)\varepsilon_{\ell_3} \tag{34}$$

As shown in Equation (34), the floating-point number computed by this multiplication is $\uparrow_\circ (f_1 f_2)$. The initial first-order errors $\omega_1^{\ell_1}\varepsilon_{\ell_1}$ and $\omega_2^{\ell_2}\varepsilon_{\ell_2}$ are multiplied by f_2 and f_1 respectively. ν denotes the empty word. In addition, the multiplication introduces a new first-order error $\downarrow_\circ (f_1 f_2)$ which is attached to the formal variable ε_{ℓ_3} in order to indicate that this error is due to the product occurring at the control point ℓ_3. Finally, this operation also introduces a second-order error that we attach to the formal variable ε_{hi}. In Figure 3, Equation (31) is a generalization of Equation (34). The term for division is obtained by means of a power series development.

This semantics details the contribution to the global error of the first-order error terms and globally computes the higher-order error arising during the calculation. In practice, higher-order errors are often negligible. So, this semantics allows us to determine the sources of imprecision due to first order errors while checking that the higher-order errors are actually globally negligible. With respect to the semantics $[\![.]\!]_{\mathbb{E}}$, we have the following result.

Proposition 5 *Let a be an arithmetic expression such that $[\![a]\!]_{\mathbb{E}} = f\varepsilon_f + e\varepsilon_e$ and $[\![a]\!]_{\mathbb{W}} = \omega\varepsilon + \sum_{\ell \in \mathcal{L}} \omega^\ell \varepsilon_\ell$. Then $\omega = f$ and $\sum_{\ell \in \mathcal{L}} \omega^\ell = e$.*

Like $[\![.]\!]_{\mathbb{E}}$, $[\![.]\!]_{\mathbb{W}}$ is not directly implementable since it uses real numbers and the function $\downarrow_\circ :\ \mathbb{R} \to \mathbb{R}$. However, $[\![.]\!]_{\mathbb{W}}$ can also be approximated in the same way than $[\![.]\!]_{\mathbb{E}}$. The error terms can be calculated for instance with intervals or stochastic numbers, yielding new semantics $[\![.]\!]_{\mathbb{WI}}$ and $[\![.]\!]_{\mathbb{WS}}$.

In addition, the first order error terms of $[\![.]\!]_{\mathbb{W}}$ can be related to the partial derivatives computed by $[\![.]\!]_{\mathbb{D}}$. Let d_i be a datum identified by the control point ℓ of a program p such that p computes a function $g(d_1, \ldots, d_n)$. Then the term $\downarrow_\circ (d_i) \times \frac{\partial g}{\partial d_i}(d_1, \ldots, d_n)$ linearly approximates the error term $\omega^\ell \boldsymbol{\varepsilon}_\ell$ of $[\![.]\!]_{\mathbb{W}}$. With respect to the exact semantics $[\![.]\!]_{\mathbb{W}}$, $[\![.]\!]_{\mathbb{D}}$ neglects the higher order error term $\omega^{hi}\boldsymbol{\varepsilon}_{hi}$ as well as any error term $\omega^\ell \boldsymbol{\varepsilon}_\ell$ such that ℓ is not related to an initial datum, i.e. such that ℓ is related to an operation.

Higher order errors often are negligible and are, in practice, neglected by most methods (but $[\![.]\!]_{\mathbb{W}}$ and $[\![.]\!]_{\mathbb{I}}$). The program of Section 6 is a case in which, for some inputs, first order errors are negligible while higher order errors are not: If the parameter $x = f\boldsymbol{\varepsilon} + \omega\boldsymbol{\varepsilon}_x$ is such that $f < 1$ and $f + \omega > 1$ then, in the floating-point numbers, $g(x) \to 0$ as $n \to \infty$, while in the reals, $g(x + \omega) \to \infty$ as $n \to \infty$. The global error tends towards infinity but it can be shown that all the first order error terms tend to 0. Since only the higher order errors are significant, a method neglecting them should not detect any problem in this computation.

8 Static analysis

Numerical computations are increasingly used in critical embedded systems like planes or nuclear power plants. In this area, the designers used to use fixed-point numbers for two reasons: first, the embedded processors did not have floating-point units and, secondly, the calculations were rather simple. Nowadays, floating-point numbers are more and more used in these systems, due to the increasing complexity of the calculations and because FPU are integrated in most processors.

Concerning numerical computations, the validation of a critical embedded system requires at least to prove that the precision of a variable is always acceptable. Executions can be instrumented with an automatic differentiation or a stochastic semantics. However, these methods do not enable one to have a full covering of the possible configurations and representative data sets are specially difficult to define since very close inputs may give very different results in terms of precision. In addition, for the same execution path, the precision may greatly differ, for different data sets. Static analysis addresses these problems by enabling to validate in a single time a code for a large class of inputs usually defined by ranges for all the parameters. In this article, we focus on validation of float calculations but most results are similar for fixed point calculations. Basically, in the latter case, the function $\downarrow_\circ$ must be redefined. In the following, we detail how the methods of sections 3 to 7 can be used for static analysis.

Many static analyzers implement an interval analysis (e.g. [4]). The main drawback of this approach was already outlined for the dynamic semantics: as discussed in Section 4, when an interval is large, one cannot assert that the precision of the results is acceptable. In addition, the intervals given to a static analyzer usually are larger than these used to simulate a single run. As a consequence, a static analyzer based on $[\![.]\!]_{\mathbb{I}}$ often is too pessimistic to assert the accuracy of a floating-point computation.

To our knowledge, no static analysis based on stochastic arithmetics as been defined nor experimented yet. However, as suggested in [13], we can expect interesting results from static analyses combining $[\![.]\!]_{\mathbb{S}}$ and recent work on static analysis of probabilistic semantics [25, 26].

Automatic differentiation seems a good candidate to static analysis even if the classical semantics $[\![.]\!]_{\mathbb{D}}$ has some limitations: some error terms are neglected and, for the others, there is a linear approximation. A semantics $[\![.]\!]_{\mathbb{WD}}$ was recently proposed that performs automatic differentiation behind the semantics $[\![.]\!]_{\mathbb{W}}$ of error series [23]. In $[\![.]\!]_{\mathbb{WD}}$, no error term is neglected but the linear approximation remains. For loops, a static stability test based on the calculation of abstract Lyapunov exponents [1] can be used [23]. It allows one to iterate just enough to prove that the iterates of a function related to the body of a loop are stable.

Finally, concerning error series, a static analyzer named Fluctuat and based on $[\![.]\!]_{\mathbb{WI}}$ has been implemented [14]. For a value $f\boldsymbol{\varepsilon} + \sum_{\ell \in \mathcal{L}} \omega^{\ell} \boldsymbol{\varepsilon}_{\ell}$, the float f is abstracted by a float interval and the error terms ω^{ℓ} are abstracted by intervals of multiple precision numbers. Fluctuat analyzes C programs and is currently experimented in an industrial context, for the validation of large-size avionic embedded softwares.

9 Experimental Results

In this section, we present some experimental results obtained using the methods described earlier. The tools and libraries that we use for interval arithmetics, stochastic arithmetics, automatic differentiation and error series respectively are MPFI [30], CADNA [9], ADOL-C [15] and Fluctuat [14]. Each example, written in C, was designed to illustrate how a certain method behaves in a particular case (mostly to show their limitations even if they all behave well on many other examples) but, in the following, we test all the methods for each case. Depending on the cases, Fluctuat is asked to unroll the loops or to work as a static analyzer. When unrolling the loops, the tool implements the semantics $[\![.]\!]_{\mathbb{WI}}$ discussed in Section 7. The results are given in the tables of figures 4, 5, 6 and 7. In each table, the *comment* column contains information on how the methods are configured and the *interpretation* column says what we can conclude from the test, assuming that we do not know anything else on the programs. So, if a method fails to detect a problem, the corresponding conclusion in the *interpretation* column is wrong.

Our first example is the program already introduced in Section 4 to show how difficult it is to distinguish a purely numerical error from a sensitivity error, mainly with an interval method. This example was inspired by the well-known Patriot case (see, e.g. [13]) but, here, the solutions in $\mathbb{R}$ and $\mathbb{F}$ are finite.

```
float x = 1.0; float y = 1.0e-8;
for (int i=0;i<1e8;i++) { x = x-y; }
```

	Result	Comment	Interpretation
$[\![\cdot]\!]_{\mathbb{E}}$	x=$1.0\boldsymbol{\varepsilon}_f - 1.0\boldsymbol{\varepsilon}_e$	$[\![\cdot]\!]_{\mathbb{E}}$ is the theoretical, non implementable semantics.	An absorption arises at each iteration. The float result is 1.0 and the error w.r.t. to the real solution exactly is -1.0.
MPFI	x=[-1.244141e1,1.000000]	`x` is initialized by `mpfi_init2(x,24)` to simulate the IEEE 754 Standard simple precision mode.	The real solution as well as the float solution belong to the given interval. The error may be as large as the interval width, i.e. 11.44141.
MPFI	x=[-4.11312855843084e-9,3.28835822230294e-9]	`x` is initialized by `mpfi_init_set_d` to obtain a highly accurate result.	The real solution is very close to zero. There is no unstability in this example.
CADNA	x=-0.3808, cestac= 4	`x` and `y` are declared as `single_st` numbers.	With high probability, the real solution is $\mathtt{x} = -0.3808$ and the first four significant digits seem correct.
ADOL-C	$\mathtt{x} = 1.0$, $\frac{\partial \mathtt{x}}{\partial \mathtt{y}} = -1.0e8$	Since ADOL only has double precision numbers, this test has been carried out using `adouble` numbers and $\mathtt{y} = 1.0e-22$. Results have been transposed to our example.	The float result is 1.0 but the sensitivity to `y` is high. By linear approximation, the real solution is $1.0 + \frac{\partial \mathtt{x}}{\partial \mathtt{y}} \times \Delta\mathtt{y}$ where $\Delta\mathtt{y}$ is the initial error on `y`.
Fluctuat	x=$1.0\boldsymbol{\varepsilon}_f + [-\infty,\text{-1.0e-8}]\boldsymbol{\varepsilon}_e$	No instrumentation of the code. Fluctuat does not unroll the loop (5 iterations are carried out).	Fluctuat detects that there is possibly (but not surely) a large negative error on the result.

Fig. 4. *Experimental results for the program iterating* **x=x-1.0e-8**.

Our experimental results are given in Table 4. As discussed in Section 4, the result returned by the interval method contains the interval $[0, 1]$ if MPFI is asked to simulate single precision numbers. One can conclude that both the real and float solution belong to this interval, but we cannot conclude on the nature of the inaccuracy. If MPFI is asked to use multiple precision numbers, it outputs a small interval around 0. We can conclude that the real solution is close to 0, but no hint is given about the error arising in the non-instrumented code.

Concerning the other methods, CADNA computes that the real solution is close to -0.38, which is rather imprecise. As outlined in Section 5, this probably stems from the fact that the errors arising at each iteration are not independent, as supposed by the method. ADOL-C returns the float result and indicates that it is very sensitive to the value of `y`. The pitfall is detected by the automatic differentiation library as one could expect, since the computed function is linear.

Contrarily to the first example, our second test program, taken from [13], is an unstable numerical scheme computing the n^{th} power u_n of the golden number $u_1 = \frac{\sqrt{5}-1}{2} \approx 0.618034$ using the property $u_{n+2} = u_n - u_{n+1}$.

```
double x = 1.0; double y = 0.618034;
for (i=0;i<=100;i++) {
  z=x; x=y;
  y=z-y;
};
```

Our experimental results are given in Figure 5. With MPFI, `y` is initialized to the small interval $[0.618034, 0, 618035]$. MPFI returns a large interval and we

	Result	Comment	Interpretation
MPFI	y = [-9.37805732496113e14, -1.04330402763639e13]	y initialized with `mpfi_interv_d`.	The real solution belong to the given interval. The error may be as large as the interval width, i.e. approximatively 9.27e14. The computation is unstable.
CADNA	y=-0.474119386437716e+15, cestac=15	x, y, z have `double_st` type. y was initialized to the median value of the interval.	With high probability, the real solution is y≈-0.47e+15 and the first fifteen significant digits are correct. This computation seems stable.
ADOL-C	y=-4.74119e+14, $grad$(y)=-3.54225e+20	x, y, z have `adouble` type. y is initialized to the median value of the interval.	The gradient indicates that this computation is unstable.
Fluctuat	y=[-9.37805732496110e14,-1.04330402763639e13]ε_f+[-25363.4,25363.4]ε_e	An assertion is used to initialize y to [0.618034,0.618035]$\varepsilon_f + 0\varepsilon_e$. Fluctuat is asked to unroll the loop.	The results belong to a large interval but the errors never are greater than 25363.4. If the initial error on y is null, the computation is unstable but roundoff errors are negligible.
Fluctuat	y=[-1.04330402763639e13,-1.04330402763639e13]ε_f+[-9.27373e+14,-0.00149523]ε_e	An assertion is used to initialize y to 0.618034ε_f+[0,0,02]ε_e. Fluctuat is asked to unroll the loop.	This program is very sensitive to the initial value of y.

Fig. 5. *Experimental results for the program computing the n^{th} power of the golden number. Note that in the reals $u_n \to 0$ as $n \to \infty$ if $u_1 = \frac{\sqrt{5}-1}{2}$.*

can conclude that the scheme is unstable. When Fluctuat is initialized with y=$[0.618034, 0, 618035]\varepsilon_f + 0\varepsilon_e$, which means that the initial float is an exact value belonging to the given interval, Fluctuat states that the result belongs to a wide interval but that the error term is small. So this computation is unstable but it is not much perturbed by the roundoff of the operations arising in the loop. When Fluctuat is initialized with y $= 0.618034\varepsilon_f + [0, 0.02]\varepsilon_e$, which means that the initial float exactly is $0,618034$ and that a roundoff error is attached to it, the tool detects a large sensitivity to y.

Concerning the other methods, CADNA does not detect the instability while ADOL-C does (again, this scheme is linear).

ADOL-C caught the numerical problems arising in the first two examples involving linear calculations. However, as discussed in Section 6, automatic differentiation methods may fail to detect such numerical errors, e.g. in the non-linear calculation introduced in Section 6 and repeated below:

```
  x = 1.0; y = 0.99;
  for (int i=0;i<1000;i++) { x = x*y; }
```

The experimental results obtained for this example are given in Figure 6. ADOL-C numerically computes the derivative given in Section 6 and does not detect the sensitivity of this code to the value of y. CADNA also indicates that this computation does not seem to be sensitive to its input values, possibly because the dominant error is non-linear. Using an assertion stating that initially y= $0.99\varepsilon_f + [0, 0.02]\varepsilon_e$, Fluctuat states that the float computed by a non-

	Result	Comment	Interpretation
$[\![\cdot]\!]_{\mathbb{E}}$	x= 0.43e-4$\boldsymbol{\varepsilon}_f - \omega\boldsymbol{\varepsilon}_e$	ω is scriptsize if initially y=0.99$\boldsymbol{\varepsilon}_f + \omega'\boldsymbol{\varepsilon}_e$ with $\omega' < 0.01$. ω is large otherwise.	This computation is sensitive to y.
MPFI	x=[4.31712474106544e-5,4.31712474106612e-5]	x and y initialized by `mpfi_init_set_d`.	The float solution is close to the real one since the interval width is small.
CADNA	x=0.43171247410657e-4, cestac =14	x and y are declared as `double_st` numbers.	With high probability, the real solution is x $\approx 0.43e-4$ and the first fourteen significant digits are correct.
ADOL-C	x=4.31712e-5, $\frac{\partial x}{\partial y}$=0.0436073	The experiment has been carried out using `adouble` numbers.	The derivative is small. This computation seems to be not much sensitive to y.
Fluctuat	x=4.31712474106578e-5$\boldsymbol{\varepsilon}_f$+[8.22526e-21,6.17629e-20]$\boldsymbol{\varepsilon}_e$	No instrumentation of the code. Fluctuat is asked to fully unroll the loop.	Assuming that initially y is exact, no numerical error arises in this execution.
Fluctuat	x=4.31712474106578e-5$\boldsymbol{\varepsilon}_f$+[-1.42622e-12,20959.2]$\boldsymbol{\varepsilon}_e$	An assertion states that initially y = 0.99$\boldsymbol{\varepsilon}_f$ + $[0.0, 0.02]\boldsymbol{\varepsilon}_e$. Fluctuat is asked to fully unroll the loop. The tool states that the dominant error is a higher order error.	The sensitivity to y is detected.

Fig. 6. *Experimental results for the non-linear computation of Section 6.*

instrumented version of the program goes to 0 while the error with respect to the real result is increasingly large.

We end this section with a case study taken from [22] and concerning the validation of a class of executions of a simple numerical program implementating Jacobi's iterative method to solve a system of linear equations. As discussed in Section 8, in order to validate a class of executions for this program, we aim at showing that the errors arising during any execution performed with parameters taken in a certain set remain acceptable. We consider the systems:

$$(S_1)\ :\ \begin{cases} 2x + y = \frac{5}{3} \\ x + 3y = \frac{5}{2} \end{cases} \qquad (S_2)\ :\ \begin{cases} x_{n+1} = \frac{5}{6} - \frac{1}{2}y_n \\ y_{n+1} = \frac{5}{6} - \frac{1}{3}x_n \end{cases}$$

$$(S_3)\ :\ \begin{cases} x_{n+1} = [0.80, 0.85] - [0.4, 0.6]y_n \\ y_{n+1} = [0.80, 0.85] - [0.30, 0.35]x_n \end{cases}$$

To solve (S_1) by Jacobi's method [12], the sequence (S_2) is computed. (S_3) defines a class of systems including (S_2). Any system taken in the ranges given in (S_3) is stable The program implementing (S_2) is given below. The initial values are $x_0 = y_0 = [2.0, 3.0]$.

```
int i; double x1,y1;
double a = [0.8,0.85];  double b = [0.4,0.6];
double c = [0.8,0.85];  double d = [0.3,0.35];
double x2 = [2.0,3.0];  double y2 = [2.0,3.0];
  for(i=0;i<1000;i++) {
  x1 = x2; y1 = y2;
  x2 = a-b*y1;
  y2 = c-d*x1;
}
```

	Result	Comment	Interpretation
MPFI	x=[3.53658536585365e-1,6.16279069767442e-1], y=[5.84302325581395e-1,7.43902439024391e-1]	`x` and `y` initialized by `mpfi_interv_d`.	The real solution as well as the float solution belong to the given intervals. The errors never are larger than the interval widths, i.e. about 0.26 for `x` and 0.15 for `y`.
CADNA	x=0.49253731343283, cestac=15; y=0.664925373134328, cestac=15	`a`, `b`, `c`, `d`, `x2` and `y2` are initialized to the median values of the intervals using the `double_st` type.	The large number of common digits indicates that the program, executed with the chosen parameters, is stable, with high probability.
ADOL-C	x=0.492537, *grad*(x)=0.0971263, y=0.664925, *grad*(y)=0.475897	`a`, `b`, `c`, `d`, `x2` and `y2` are initialized to the median values of the intervals using the `adouble` type.	The gradients indicate that this computation is stable in the neighborhood of the chosen parameters.
Fluctuat	x=[3.53658536585366e-1,6.16279069767442e-1]$\boldsymbol{\varepsilon}_f$+[-1.58101e-16,1.58101e-16]$\boldsymbol{\varepsilon}_e$ y=[5.84302325581395e-1,7.43902439024390e-1]$\boldsymbol{\varepsilon}_f$+[-1.24724e-16,1.24724e-16]$\boldsymbol{\varepsilon}_e$	Assertions are used to initialize the identifiers. Fluctuat is asked to unroll the loop.	Fluctuat states that the errors on `x` and `y` never are larger than, approximatively, $1.0e-16$ for any execution.

Fig. 7. *Experimental results for the program implementing Jacobi's Method.*

Our results are given in Figure 7. MPFI outputs small intervals enabling to assess the stability of the class of executions: the errors on `x` and `y` never exceed 0.26 and 0.15, respectively. Fluctuat finds intervals for `x` and `y` comparable to these of MPFI and, additionally, states that for any execution, the errors on `x` and `y` never exceed 2.0e-16, approximatively. For a particular execution achieved using the median values of the intervals, CADNA and ADOL-C also claim that the computation is stable. However, this does not enable us to conclude on the stability of the whole class of executions. It is interesting to note that, in this test, all the methods output comparable values: the real number output by CADNA belongs to MPFI and Fluctuat intervals which are almost identical. The error term of Fluctuat (≈1e-16) is in adequacy with CADNA result (`cestac`=15).

10 Conclusion

The validation of the numerical quality of programs is a difficult research topic. Independently of any tool or method, that is independently of *how* the validation can be carried out, the properties that must be proven, i.e. *what* must be verified, breads many discussions. In addition to the accuracy losses introduced by floating-point numbers, other sources of imprecision are introduced by modeling, by the choice of algorithms, etc. For instance, should an unstable numerical scheme such as the golden number example of Section 9 be considered as acceptable? On one hand the errors introduced by floating-point numbers are negligible and the program mimicks closely what happens in the reals. On the other hand the sensitivity to the possibly approximative initial value is high.

In this article, we attempted to clarify what properties some techniques exactly compute and what we can conclude from these properties about the numerical quality of the tested programs. The differences can be subtle: for instance, as discussed in sections 4 and 9, an interval method working with the same precision as the non-instrumented code does not only less accurately compute the same properties as an interval method using a higher precision. By examining closely what is computed by each technique, we do not provide sufficient conditions for the validation of numerical codes in general, as asked in the previous paragraph, but we clearly define which aspects of the whole validation can be addressed by each method.

In Section 9, we introduced academic examples containing different kinds of numerical errors and mostly designed to show the limits of each method. For the sake of simplicity, these examples are very short but all the tools tested in this article are applicable to larger, non toy programs. The fact that, in our tests, some method fails to detect a numerical precision problem does not mean that it should be depreciated: our examples have been designed to this aim, many successful applications of each tool being available in its bibliography.

References

1. K. T. Alligood, T. D. Sauer, and J. A. Yorke. *Chaos, an Introduction to Dynamical Systems.* Springer-Verlag, 1996.
2. ANSI/IEEE. *IEEE Standard for Binary Floating Point Arithmetic*, Std 754 edition, 1985.
3. Christian Bischof, Paul D. Hovland, and Boyana Norris. Implementation of automatic differentiation tools. In *Proceedings of the ACM-SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Transformations, PEPM'02.* ACM Press, 2002.
4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation, PLDI'03.* ACM Press, 2003.
5. S. Boldo and M. Daumas. Representable correcting terms for possibly underflowing floating point operations. In J.-C. Bajard and M. Schulte, editors, *Proceedings of the 16th Symposium on Computer Arithmetic.* IEEE Press, 2003.
6. H. Bronnimann and G. Melquiond. The boost interval arithmetic library. In *Proceedings of the Real Numbers and Computers Conference, RNC'5*, 2003.
7. CADNA for C/C++ source codes User's Guide. `http://www-anp.lip6.fr/cadna/Documentation/Accueil.php`.
8. F. Chaitin-Chatelin and V. Frayssé. *Lectures on Finite Precision Computations.* SIAM, 1996.
9. J.-M. Chesneaux. L'arithmétique stochastique et le logiciel CADNA. Habilitation à diriger des recherches, Université Pierre et Marie Curie, Paris, 1995.
10. M. Daumas, Rideau L., and L Théry. A generic library for floating-point numbers and its application to exact computing. In *TPHOLs'01, International Conference on Theorem Proving and Higher Order Logics*, number 2152 in LNCS. Springer-Verlag, 2001.
11. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1), 1991.
12. G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 2d edition, 1990.

13. E. Goubault. Static analyses of the precision of floating-point operations. In *Static Analysis Symposium, SAS'01*, LNCS. Springer-Verlag, 2001.
14. E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. In *European Symposium on Programming, ESOP'02*, LNCS. Springer-Verlag, 2002.
15. A. Griewank, D. Juedes, and J. Utke. ADOL-C, a package for the automatic differentiation of algorithms written in c/c++. *ACM Trans. Math. Software*, 22:131–167, 1996.
16. Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Frontiers in Applied Mathematics. SIAM, 2000.
17. M. Grimmer, K. Petras, and N. Revol. Multiple precision interval packages: Comparing different approaches. In *Proceedings of the Dagstuhl Seminar on Numerical Software with Result Verification*, LNCS. Springer-Verlag, 2003. to appear.
18. G. Hanrot, V. Lefevre, Rouillier F., and P. Zimmermann. The MPFR library. Institut de Recherche en Informatique et Automatique, 2001.
19. J. Harrison. A machine-checked theory of floating point arithmetic. In *TPHOLs'99, International Conference on Theorem Proving and Higher Order Logics*, number 1690 in LNCS. Springer-Verlag, 1999.
20. P. Langlois. Automatic linear correction of rounding errors. *BIT, Numerical Mathematics*, 41(3):515–539, 2001.
21. P. Langlois and F. Nativel. Improving automatic reduction of round-off errors. In *IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, volume 2, 1997.
22. M. Martel. Propagation of roundoff errors in finite precision computations: a semantics approach. In *European Symposium on Programming, ESOP'02*, number 2305 in LNCS. Springer-Verlag, 2002.
23. M. Martel. Static analysis of the numerical stability of loops. In *Static Analysis Symposium, SAS'02*, number 2477 in LNCS. Springer-Verlag, 2002.
24. C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating-point numbers. In *CP'2001, Seventh International Conference on Principles and Practice of Constraint Programming*, number 2239 in LNCS. Springer-Verlag, 2001.
25. D. Monniaux. Abstract interpretation of probabilistic semantics. In *Static Analysis Symposium, SAS'00*, number 1824 in LNCS. Springer-Verlag, 2000.
26. D. Monniaux. An abstract Monte-Carlo method for the analysis of probabilistic programs. In *ACM Symposium on Principles of Programming Languages, POPL'01*. ACM Press, 2001.
27. R. E. Moore. *Methods and Applications of Interval Analysis*. SIAM, 1979.
28. P. J. Potts, A. Edalat, and H. M. Escardó. Semantics of exact real arithmetic. In *Procs of Logic in Computer Science*. IEEE Computer Society Press, 1997.
29. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–144. IEEE Computer Society Press, 1991.
30. N. Revol and F. Rouillier. Motivations for an arbitrary precision interval arithmetic and the MPFI library. Technical Report RR-200227, Laboratoire de l'Informatique du Parallélisme, ENS-Lyon, France, 2002.
31. J. Vignes. A stochastic arithmetic for reliable scientific computation. *Mathematics and Computers in Simulation*, 35(3):233–261, 1993.