Rigorous Enclosure of Round-Off Errors in Floating-Point Computations*

Rémy Garcia, Claude Michel, and Michel Rueher

Université Côte d'Azur, CNRS, I3S, France
firstname.lastname@i3s.unice.fr

Abstract. Efficient tools for error analysis of programs with floating-point computations are available. Most of them provide an over-approximation of the floating-point errors. The point is that these approximations are often too coarse to evaluate the effective impact of the error on the behaviour of a program. Some of these tools compute an under-approximation of the maximal error. But, these under-approximations are either not rigorous or not reachable. In this paper, we introduce a new approach to rigorously enclose the maximal error by means of an over-approximation of the error and an under-approximation computed by means of rational arithmetic. Moreover, our system, called FErA, provides input values that exercise the under-approximations. We outline the advantages and limits of our framework and compare our approach with state-of-the-art methods for over-approximating errors as well as the ones computing under-approximation of the maximal error. Preliminary experiments on standard benchmarks are promising. FErA not only computes good error bounds on most benchmarks but also provides an effective lower bound on the maximal error.

Keywords: floating-point numbers · round-off error · constraints over floating-point numbers · optimization

1 Introduction

Floating-point computations involve errors due to rounding operations that characterize the distance between the intended computations over the reals and the actual computations over the floats. An error occurs at the level of each basic operation when its result is rounded to the nearest representable floating-point number. The final error results from the combination of the rounding errors produced by each basic operation involved in an expression and some initial errors linked to input variables and constants. Such errors impact the precision and the stability of computations and can lead to an execution path over the floats that is significantly different from the expected path over the reals. A faithfull account of these errors is mandatory to capture the actual behaviour of critical programs with floating-point computations.

* This work was partially supported by ANR COVERIF (ANR–15–CE25–0002)

Efficient tools exist for error analysis that rely on an over-approximation of the errors in programs with floating-point computations. For instance, Fluctuat [11, 10] is an abstract interpreter that combines affine arithmetic and zonotopes to analyze the robustness of programs over the floats, FPTaylor [24, 23] uses symbolic Taylor expansions to compute tight bounds of the error, and PRECiSA [21, 27] is a more recent tool that relies on static analysis. Other tools compute an under-approximation of errors to find a lower bound of the maximal absolute error, e.g., FPSDP [16] which relies on semidefinite programming or, S3FP [2] that uses guided random testing to find inputs causing the worst error. Over-approximations and under-approximations of errors are complementary approaches for providing better enclosures of the maximal error. However, none of the available tools compute both an over-approximation and an under-approximation of errors. Such an enclosure would be very useful to give insights on the maximal absolute error, and how far computed bounds are from it. It is important to outline that approximations do not capture the effective behaviour of a program: they may generate *false positives*, that is to say, report that an assertion might be violated even so in practice none of the input values can exercise the related case. To get rid of *false positives*, computing maximal errors, i.e., the greatest reachable absolute errors, is required. Providing an enclosure of the maximal error, and even finding it, is the goal of the work presented in this paper.

The kernel of our approach is a branch-and-bound algorithm that not only provides an upper bound of the maximal error of a program with floating-point operations but also a reachable error exercised by computed input values. This is a key issue in real problems. This branch-and-bound algorithm is embedded in a solver over the floats [30, 18, 1, 19, 20] extended to constraints over errors [9]. The resulting system, called FErA (**F**loating-point **Er**ror **A**nalyzer), provides not only a sound over-approximation of the maximal error but also a reachable under-approximation with input values that exercise it. FErA uses rational arithmetic because we want to provide a correct lower and upper bound of the errors. A consequence of this choice is that FErA only handles basic arithmetic operators (namely $+, -, \times, /$). To our knowledge, our tool is the first one that combines upper and lower bounding of maximal round-off errors. A key point of FErA is that both bounds relies on each other for improvement.

Maximizing an error can be very expensive because the errors are unevenly distributed. Even on a single operation, such a distribution is cumbersome and finding input values that exercise it often resort to an enumeration process. A combination of floating-point operations often worsen this behaviour, but may, in some cases, soften it thanks to error compensations. One advantage of our approach is that the branch-and-bound is an anytime algorithm, i.e., it always provides an enclosure of the maximal error alongside input values exercising the lower bound.

```
z = (3*x+y)/w;

if (z - 10 <= δ) {
  proceed();
} else {
  raiseAlarm();
}
```

Example 1. Simple program

1.1 Motivating example

Consider the piece of code in Example 1 that computes $z = (3 * x + y)/w$ using 64-bit floating-point numbers with $x \in [7, 9]$, $y \in [3, 5]$, and $w \in [2, 4]$.
The computation of z precedes a typical condition of a control-command code where the then-branch is activated when $z <= 10$. When $z - 10$ is lower than a given tolerance δ, values supported by z are considered as safe and related computations can be done. Otherwise, an alarm must be raised. As $z-10$ satisfies Sterbenz property [25] when $z \approx 10$, only the error on the computation of z impacts the conditional. Such a conditional is typically known as an *unstable test*, where the flow over $\mathbb{F}$ can diverge from the one over $\mathbb{R}$ [28]. Now, assume that δ is set to $5.32e{-}15$. The issue is to know whether this piece of code behaves as expected, i.e., to know whether the error on z is small enough to avoid raising the alarm when the value of z is less than or equal to 10 on real numbers.

	FPTaylor	PRECiSA	Fluctuat	FErA
Example 1	5.15e-15	5.08e-15	6.28e-15	4.96e-15

Table 1. Absolute error bound

Table 1 reports the error values given by FPTaylor [24, 23], PRECiSA [21, 27], Fluctuat [11, 10], and our tool FErA. Fluctuat compute a bound greater than δ whereas FErA, FPTaylor, and PRECiSA compute a bound lower than δ. Results from Fluctuat suggest that the alarm might inappropriately be raised.
FErA computes a round-off error upper bound of $4.96e{-}15$ in about 0.185 seconds. It also computes a lower bound on the largest absolute error of $3.55e{-}15$ exercised by the following input values:

$$
\begin{aligned}
x &= 8.99999999999996624922 & e_x &= -8.88178419700125232339e{-}16 \\
y &= 4.99999999999994848565 & e_y &= -4.44089209850062616169e{-}16 \\
w &= 3.19999999999998419042 & e_w &= +2.22044604925031308085e{-}16 \\
z &= 10.0000000000000035527 & e_z &= -3.55271367880050092936e{-}15
\end{aligned}
$$

```
z = (3*x+y)/w;

if(z <= 10 + δ) {
  proceed();
} else {
  proceedWithError();
}
```

Example 2. Simple program without alarm

In other words, our sound optimizer not only guarantee that errors can never raise an alarm, but it also provides an enclosure of the largest absolute error. Such an enclosure having a ratio of 1.4 shows that the round-off error bound of FErA is close to the actual errors.
Now, replace the piece of code by the one from Example 2 and assume δ is set to $3.55e{-}15$. The only change is that no alarm is raised when the error is greater than δ but the procedure proceedWithError is activated.
The issue here is to know if there exist at least one case where proceedWithError is reached when z is less than or equal to 10 on real numbers. FErA computes an enclosure on the largest absolute error of $[3.55e{-}15, 4.96e{-}15]$ with input values exercising the lower bound. So, it ensures that there exist at least one case when proceedWithError is taken with an error bigger than δ. FPTaylor, PRECiSA, and Fluctuat are unable to do so as none of them compute a reachable lower bound on the largest absolute error.

The rest of the paper is organized as follows: Section 2 introduces notations and definitions. Section 3 recalls the constraint system for round-off error analysis and explains how the filtering works. Section 4 formally introduces the branch-and-bound algorithm and its main properties. Section 5 describes in more detail related works for computing a lower bound on the maximal error and their pitfalls. Section 6 illustrates the capabilities of FErA on well-known examples and provides preliminary experiments on a set of standard benchmarks.

2 Notation and definitions

Our system for round-off error analysis focuses on the four classical arithmetic operations: $+, -, \times, /$ for which the error can be computed exactly using rational numbers [9]. As usual, a constraint satisfaction problem, or CSP, is defined by a triple $\langle X, D, C\rangle$, where X denotes the set of variables, D, the set of domains, and C, the set of constraints. The set of floating-point numbers is denoted $\mathbb{F}$, the set of rational numbers is denoted $\mathbb{Q}$, and the set of real numbers is denoted $\mathbb{R}$. For each floating-point variable x in X, the domain of values of x is represented by the interval $\mathbf{x} = [\underline{\mathbf{x}}, \overline{\mathbf{x}}] = \{x \in \mathbb{F} \mid \ \underline{\mathbf{x}} \leq x \leq \overline{\mathbf{x}}\}$, where $\underline{\mathbf{x}}$ (resp. $\overline{\mathbf{x}}$) denotes the interval lower (resp. upper) bound, while the domain of errors of x is represented by the interval $\mathbf{e_x} = [\underline{\mathbf{e_x}}, \overline{\mathbf{e_x}}] = \{e_x \in \mathbb{Q} \mid \ \underline{\mathbf{e_x}} \leq e_x \leq \overline{\mathbf{e_x}}\}$. $\mathbf{x}_{\mathbb{F}}$ (respectively, $\mathbf{x}_{\mathbb{Q}}$

and $\mathbf{x}_{\mathbb{R}}$) denotes a variable that takes its values in $\mathbb{F}$ (respectively, $\mathbb{Q}$ and $\mathbb{R}$). A variable is instantiated when its domain of values is reduced to a degenerate interval, i.e., a singleton.

3 A constraint system for round-off error

The branch-and-bound algorithm at the kernel of our framework is based on a constraint system on errors that we briefly introduce in this section.

3.1 Computing rounding errors

The IEEE 754 standard [14] requires correct rounding for the four basic operations of floating-point arithmetic. The result of such an operation over the floats must be equal to the rounding of the result of the equivalent operation over the reals. More formally, $z = x \odot y = round(x \cdot y)$ where z, x, and y are floating-point numbers, $\odot$ is one of the four basic arithmetic operations on floats, namely, $\oplus$, $\ominus$, $\otimes$, $\oslash$, while $\cdot$ are the equivalent operations on reals, namely, $+$, $-$, $\times$, $/$; *round* being the rounding function. This property is used to bound the error introduced by each elementary operation on floats by $\pm\frac{1}{2}ulp(z)$[1] when the rounding mode is set to round to the "nearest even" float, the most frequently used rounding mode.
The deviation of a computation over the floats takes root in each elementary operation. So, it is possible to rebuild the final deviation of an expression from the composition of errors due to each elementary operation involved in that expression. Let us consider a simple operation like the subtraction as in $z = x \ominus y$: input variables, x and y, can come with errors attached due to previous computations. For instance, the deviation on the computation of x, e_x, is given by $e_x = x_{\mathbb{R}} - x_{\mathbb{F}}$ where $x_{\mathbb{R}}$ and $x_{\mathbb{F}}$ denote the expected results, respectively, on reals and on floats.
The computation deviation due to a subtraction can be formulated as follows: for $z = x \ominus y$, e_z, the error on z, is equal to $(x_{\mathbb{R}} - y_{\mathbb{R}}) - (x_{\mathbb{F}} \ominus y_{\mathbb{F}})$.
As $e_x = x_{\mathbb{R}} - x_{\mathbb{F}}$ and $e_y = y_{\mathbb{R}} - y_{\mathbb{F}}$, we have

$$e_z = ((x_{\mathbb{F}} + e_x) - (y_{\mathbb{F}} + e_y)) - (x_{\mathbb{F}} \ominus y_{\mathbb{F}})$$

So, the deviation between the result on reals and the result on floats for a subtraction can be computed by the following formula:

$$e_z = e_x - e_y + ((x_{\mathbb{F}} - y_{\mathbb{F}}) - (x_{\mathbb{F}} \ominus y_{\mathbb{F}}))$$

where $(x_{\mathbb{F}} - y_{\mathbb{F}}) - (x_{\mathbb{F}} \ominus y_{\mathbb{F}})$ characterizes the error produced by the subtraction operation itself. Lets $e_{\ominus}$ denotes the error produced by the subtraction operation. The formula can then be denoted by:

$$e_z = e_x - e_y + e_{\ominus}$$

$$\begin{aligned}
\text{Addition: } z = x \oplus y \rightarrow e_z &= e_x + e_y + e_\oplus \\
\text{Subtraction: } z = x \ominus y \rightarrow e_z &= e_x - e_y + e_\ominus \\
\text{Multiplication: } z = x \otimes y \rightarrow e_z &= x_\mathbb{F} e_y + y_\mathbb{F} e_x + e_x e_y + e_\otimes \\
\text{Division: } z = x \oslash y \rightarrow e_z &= \frac{y_\mathbb{F} e_x - x_\mathbb{F} e_y}{y_\mathbb{F}(y_\mathbb{F} + e_y)} + e_\oslash
\end{aligned}$$

Fig. 1. Computation of deviation for basic operations

that combines deviations from input values and the deviation introduced by the elementary operation.
Computation deviations for all four basic operations are given in Figure 1. For each of these formulae, the error computation combines deviations from input values and the error introduced by the current operation. Note that, for the multiplication and division, this deviation is proportional to the input values.
All these formulae compute the difference between the expected result on reals and the actual one on floats for a basic operation. Our constraint solver over the errors relies on these formulae.

3.2 A CSP with three domains

As in a classical *CSP*, to each variable x is associated $\mathbf{x}$ its domain of values. The domain $\mathbf{x}$ denotes the set of possible values that this variable can take. When the variable takes values in $\mathbb{F}$, its domain of values is represented by an interval of floats:

$$\mathbf{x}_\mathbb{F} = [\underline{\mathbf{x}}_\mathbb{F}, \overline{\mathbf{x}}_\mathbb{F}] = \{x_\mathbb{F} \in \mathbb{F}, \underline{\mathbf{x}}_\mathbb{F} \leq x_\mathbb{F} \leq \overline{\mathbf{x}}_\mathbb{F}\}$$

where $\underline{\mathbf{x}}_\mathbb{F} \in \mathbb{F}$ and $\overline{\mathbf{x}}_\mathbb{F} \in \mathbb{F}$
Errors require a specific domain associated with each variable of a problem. Since the arithmetic constraints processed here are reduced to the four basic operations, and since those four operations are applied on floats, i.e., a finite subset of rationals, this domain can be defined as an interval of rationals with bounds in $\mathbb{Q}$:

$$\mathbf{e}_x = [\underline{\mathbf{e}}_x, \overline{\mathbf{e}}_x] = \{e_x \in \mathbb{Q}, \underline{\mathbf{e}}_x \leq e_x \leq \overline{\mathbf{e}}_x\}$$

where $\underline{\mathbf{e}}_x \in \mathbb{Q}$ and $\overline{\mathbf{e}}_x \in \mathbb{Q}$.
The domain of errors on operations, denoted by $e_\odot$, that appears in the computation of deviations (see Figure 1) is associated with each *instance* of an arithmetic operation of a problem.

3.3 Projection functions

The filtering process of FErA is based on classical projection functions that reduce the domains of the variables. Domains of values can be reduced by means

[1] $ulp(z)$ is the distance between z and its successor (noted z^+).

Addition

$$\begin{aligned}
\mathbf{e}_z &\leftarrow \mathbf{e}_z \cap (\mathbf{e}_x + \mathbf{e}_y + \mathbf{e}_\oplus)\\
\mathbf{e}_x &\leftarrow \mathbf{e}_x \cap (\mathbf{e}_z - \mathbf{e}_y - \mathbf{e}_\oplus)\\
\mathbf{e}_y &\leftarrow \mathbf{e}_y \cap (\mathbf{e}_z - \mathbf{e}_x - \mathbf{e}_\oplus)\\
\mathbf{e}_\oplus &\leftarrow \mathbf{e}_\oplus \cap (\mathbf{e}_z - \mathbf{e}_x - \mathbf{e}_y)
\end{aligned}$$

Subtraction

$$\begin{aligned}
\mathbf{e}_z &\leftarrow \mathbf{e}_z \cap (\mathbf{e}_x - \mathbf{e}_y + \mathbf{e}_\ominus)\\
\mathbf{e}_x &\leftarrow \mathbf{e}_x \cap (\mathbf{e}_z + \mathbf{e}_y - \mathbf{e}_\ominus)\\
\mathbf{e}_y &\leftarrow \mathbf{e}_y \cap (-\mathbf{e}_z + \mathbf{e}_x + \mathbf{e}_\ominus)\\
\mathbf{e}_\ominus &\leftarrow \mathbf{e}_\ominus \cap (\mathbf{e}_z - \mathbf{e}_x + \mathbf{e}_y)
\end{aligned}$$

Multiplication

$$\begin{aligned}
\mathbf{e}_z &\leftarrow \mathbf{e}_z \cap (\mathbf{x}_\mathbb{F}\mathbf{e}_y + \mathbf{y}_\mathbb{F}\mathbf{e}_x + \mathbf{e}_x\mathbf{e}_y + \mathbf{e}_\otimes)\\
\mathbf{e}_x &\leftarrow \mathbf{e}_x \cap \left(\frac{\mathbf{e}_z - \mathbf{x}_\mathbb{F}\mathbf{e}_y - \mathbf{e}_\otimes}{\mathbf{y}_\mathbb{F} + \mathbf{e}_y}\right)\\
\mathbf{e}_y &\leftarrow \mathbf{e}_y \cap \left(\frac{\mathbf{e}_z - \mathbf{y}_\mathbb{F}\mathbf{e}_x - \mathbf{e}_\otimes}{\mathbf{x}_\mathbb{F} + \mathbf{e}_x}\right)\\
\mathbf{e}_\otimes &\leftarrow \mathbf{e}_\otimes \cap (\mathbf{e}_z - \mathbf{x}_\mathbb{F}\mathbf{e}_y - \mathbf{y}_\mathbb{F}\mathbf{e}_x - \mathbf{e}_x\mathbf{e}_y)\\
\mathbf{x}_\mathbb{F} &\leftarrow \mathbf{x}_\mathbb{F} \cap \left(\frac{\mathbf{e}_z - \mathbf{y}_\mathbb{F}\mathbf{e}_x - \mathbf{e}_x\mathbf{e}_y - \mathbf{e}_\otimes}{\mathbf{e}_y}\right)\\
\mathbf{y}_\mathbb{F} &\leftarrow \mathbf{y}_\mathbb{F} \cap \left(\frac{\mathbf{e}_z - \mathbf{x}_\mathbb{F}\mathbf{e}_y - \mathbf{e}_x\mathbf{e}_y - \mathbf{e}_\otimes}{\mathbf{e}_x}\right)
\end{aligned}$$

Division

$$\begin{aligned}
\mathbf{e}_z &\leftarrow \mathbf{e}_z \cap \left(\frac{\mathbf{y}_\mathbb{F}\mathbf{e}_x - \mathbf{x}_\mathbb{F}\mathbf{e}_y}{\mathbf{y}_\mathbb{F}(\mathbf{y}_\mathbb{F} + \mathbf{e}_y)} + \mathbf{e}_\oslash\right)\\
\mathbf{e}_x &\leftarrow \mathbf{e}_x \cap \left((\mathbf{e}_z - \mathbf{e}_\oslash)(\mathbf{y}_\mathbb{F} + \mathbf{e}_y) + \frac{\mathbf{x}_\mathbb{F}\mathbf{e}_y}{\mathbf{y}_\mathbb{F}}\right)\\
\mathbf{e}_y &\leftarrow \mathbf{e}_y \cap \left(\frac{\mathbf{e}_x - \mathbf{e}_z\mathbf{y}_\mathbb{F} + \mathbf{e}_\oslash\mathbf{y}_\mathbb{F}}{\mathbf{e}_z - \mathbf{e}_\oslash + \frac{\mathbf{x}_\mathbb{F}}{\mathbf{y}_\mathbb{F}}}\right)\\
\mathbf{e}_\oslash &\leftarrow \mathbf{e}_\oslash \cap \left(\mathbf{e}_z - \frac{\mathbf{y}_\mathbb{F}\mathbf{e}_x - \mathbf{x}_\mathbb{F}\mathbf{e}_y}{\mathbf{y}_\mathbb{F}(\mathbf{y}_\mathbb{F} + \mathbf{e}_y)}\right)\\
\mathbf{x}_\mathbb{F} &\leftarrow \mathbf{x}_\mathbb{F} \cap \left(\frac{(\mathbf{e}_\oslash - \mathbf{e}_z)\mathbf{y}_\mathbb{F}(\mathbf{y}_\mathbb{F} + \mathbf{e}_y) + \mathbf{y}_\mathbb{F}\mathbf{e}_x}{\mathbf{e}_y}\right)\\
\mathbf{y}_\mathbb{F} &\leftarrow \mathbf{y}_\mathbb{F} \cap [\min(\underline{\delta}_1, \underline{\delta}_2), \max(\overline{\delta}_1, \overline{\delta}_2)]
\end{aligned}$$

with

$$\delta_1 \leftarrow \frac{\mathbf{e}_x - (\mathbf{e}_z - \mathbf{e}_\oslash)\mathbf{e}_y - \sqrt{\mathbf{\Delta}}}{2(\mathbf{e}_z - \mathbf{e}_\oslash)} \qquad \delta_2 \leftarrow \frac{\mathbf{e}_x - (\mathbf{e}_z - \mathbf{e}_\oslash)\mathbf{e}_y + \sqrt{\mathbf{\Delta}}}{2(\mathbf{e}_z - \mathbf{e}_\oslash)}$$

$$\mathbf{\Delta} \leftarrow [0, +\infty) \cap ((\mathbf{e}_z - \mathbf{e}_\oslash)\mathbf{e}_y - \mathbf{e}_x)^2 + 4(\mathbf{e}_z - \mathbf{e}_\oslash)\mathbf{e}_y\mathbf{x}_\mathbb{F}$$

Fig. 2. Projection functions of arithmetic operation

of standard floating-point projection functions defined in [19] and extended in [1, 18]. However, dedicated projections are required to reduce domains of errors.

The projections on the domains of errors are defined through an extension over intervals of the formulae of Figure 1. Since these formulae are written on reals, they can naturally be extended to intervals. The projections functions for the four basic arithmetic operations are detailed in Figure 2. Since no error is involved in comparison operators, their projection functions only handle domains of values. So, projection functions on the domain of errors support only arithmetic operations and assignment, where the computation error from the expression is transmitted to the assigned variable. All these projection functions are used to reduce the domains of the variables until a fixed point is reached. For the sake of efficiency, but also to get rid of potential slow convergences, the fixed point computation is not computed and the algorithm stops when domain reduction is lower than 5%.

3.4 Links between domains of values and domains of errors

Strong connections between the domain of values and the domain of errors are required to propagate reductions between these domains.
A first relation between the domain of values and the domain of errors on operations is based upon the IEEE 754 standard, that guarantees that basic arithmetic operations are correctly rounded.

$$\mathbf{e}_{\odot} \leftarrow \mathbf{e}_{\odot} \cap \left[-\frac{\min((\underline{z} - \underline{z}^{-}), (\overline{z} - \overline{z}^{-}))}{2}, +\frac{\max((\underline{z}^{+} - \underline{z}), (\overline{z}^{+} - \overline{z}))}{2} \right]$$

where z^- and z^+ denote respectively, the greatest floating-point number strictly inferior to z and the smallest floating-point number strictly superior to z. This equation sets a relation between the domain of values and the domain of errors on operations. More precisely, it states that operation errors can never be greater than the greatest half-ulp of the domain of values of the operation result. Note that the contrapositive of this property offers another opportunity to connect the domain of values and the domain of errors. Indeed, since the absolute value of an operation error is less than the greatest half-ulp of the domain of values of the operation result, the smallest values of the domain of the result cannot be the support of a solution if $\inf(|e_{\odot}|) > 0$. In other words, these small values near zero cannot be associated to an error on the operation big enough to be in $e_{\odot}$ domain if their half-ulp is smaller than $\inf(|e_{\odot}|)$.
Finally, these links are refined by means of other well-known properties of floating-point arithmetic like the Sterbenz property of the subtraction [25] or the Hauser property on the addition [13]. Both properties give conditions under which these operations produce exact results, the same being true for the well-known property that states that $2^k \times x$ is exactly computed, provided that no overflow occurs.

3.5 Constraints over errors

A dedicated function, $err(x)$, is used to express constraints over errors. For instance, $abs(err(x)) \geq \epsilon$, denotes a constraint on the error linked to variable x that must be, in absolute value, greater or equal to ϵ. It should be noted that since errors are taking their values in $\mathbb{Q}$, this constraint is over rationals.
Note that when a constraint mixes errors and floating-point variables, the latter are automatically promoted to rationals.

4 A branch-and-bound algorithm to maximize the error

We use a branch-and-bound algorithm (see Algorithm 1) to maximize a given absolute error from a CSP. Such an error characterizes the greatest possible deviation between the expected computation over the reals and the actual computation over the floats. Note that the algorithm can easily be changed to maximize a signed error.

The branch-and-bound algorithm takes as inputs a CSP $\langle X, C, D\rangle$, and e, an error to maximize. This error results from floating-point computations along a given path in a program. It computes a lower bound, or primal, e^*, i.e., the maximal error computed so far and an upper bound, or dual, $\overline{e}$ of the maximal error e, i.e., the current best over-approximation of the error. Primal and dual bounds the maximal error: $e^* \leq e \leq \overline{e}$. Both of those bounds are expressed in absolute value. The computed primal e^* is a reachable error exercised by computed input values. These values and the computed bounds are returned by our algorithm. S is the ordered set of couples (e, sol) where e and sol are, respectively, an error and its corresponding input values. A box B is the cartesian product of variable domains. For the sake of clarity, a box B can be used as exponent, e.g., $\mathbf{x}^B$ indicates that an element $\mathbf{x}$ is in box B. L is the set of boxes left to compute.

Stopping criteria. The primary aim of the branch-and-bound algorithm is to compute the maximal error. This is achieved when the primal is equal to the dual. However, such a condition may be difficult to meet.
A first issue comes from the dependency problem which appears on expressions with multiples occurrences. Multiple occurrences of variables is a critical issue in interval arithmetic since each occurrence of a variable is considered as a different variable with the same domain. This dependency problem results in overestimations in the evaluation of the possible values that an expression can take. For instance, let $y = x \times x$ with $x \in [-1, 1]$, classical interval arithmetic yields $[-1, 1]$ whereas the exact interval is $[0, 1]$. Such a drawback arise in projection functions for computing errors that contains multiple occurrences like in multiplication and division. It can leads to unnecessary over-approximation of resulting intervals. A direct consequence of this problem is that the dual is overestimated and therefore can not be achieve.
A second issue comes from the bounding of errors on operations by the half of an ulp. An operation error is bounded by $\frac{1}{2}\text{ulp}(z)$ where z is the result of an operation. Such a bound is highly dependent on the distribution of floating-point numbers. Consider an interval of floating-point numbers $(2^n, 2^{n+1})$, every floating-point number is separated from the next one by the same distance. In other words, every floating-point number in this interval will have the same ulp. When the domain of the result of an operation is reduced to such an interval, the bounds of $e_\odot$ are fixed and can no longer be improved by means of projection functions. This can be generalized across all operations of a CSP. Once all operation errors are fixed, then bounds cannot be tighten without enumerating values. In other words, provided that there is no multiple occurrences, the dual can no longer be lowered at this point. That is why, we stop processing a box when all the related domains are reduced to such an interval.

Box management. The algorithm manages a list L of boxes to process whose initial value is $\{B = (D, e^B \in [-\infty, +\infty])\}$ where D is the cross product of the domains of the variable as defined in the problem and e^B their associated error. It also manages the global primal and dual with $e^* = -\infty$, $\overline{e} = +\infty$ as

initial values. A box can be in three different states: *unexplored*, *discarded* or *sidelined*. *unexplored* boxes are boxes in L that still require some computations. A *discarded* box is a box whose associated error e^B is such that $\overline{e}^B \leq e^*$. In other words, such a box does not contain any error value that can improve the computation of the maximal error. It is thus removed from L. *sidelined* boxes are boxes that fullfil the property described in the stopping criteria paragraph. These boxes cannot improve maximal error computation unless if the algorithm resorts to enumeration (provided there are no multiple occurrences). As sidelined boxes are still valid boxes, the greatest over-approximation of such boxes, $\overline{e}^S$, is taken into account when updating the dual bound. Solving stops when there are no more boxes to process or when the primal e^* and the dual $\overline{e}$ are equal, i.e., when the maximal error is found.
The main loop of the branch-and-bound algorithm can be subdivided in several steps: box selection, filtering, dual updating, primal updating, and box splitting.

Box selection. We select the box B in the set L with the greatest upper bound of the error to provide more opportunities to improve both $\overline{e}$ and e^*. Indeed, the global $\overline{e}$ has its support in this box which also provides the odds of computing a better reachable error e^*. Once selected, the box B is removed from L.

Filtering. A filtering process (see Section 3), denoted Φ, is then applied to B to reduce the domains of values and the domains of errors. Note that this filtering is applied to the initial set of constraints enhanced with constraints on known bounds of e, i.e., $e^* \leq e$ and $\overline{e} \geq e$. If $\Phi(B) = \varnothing$, the selected box does not contain any solution; either because it contradicts one of the initial constraints or because of constraints over e known bounds. In both cases, the algorithm discards box B and directly jumps to the next loop iteration.

Dual update. Once B has been filtered, if the error upper bound of the current box was support of the dual $\overline{e}$ and is no longer, then $\overline{e}$ is updated. $\overline{e}$ is updated with the maximum among the upper bound of errors of the current box, of remaining boxes in L, and of sidelined boxes.

Primal update. A non empty box may contain a better primal than the current one. A *generate-and-test* procedure, `primalComputation`, attempts to compute a better one in the following way: each variable is, in turn, randomly set to a value that belongs to its domain. Note that the error distribution is such that a randomly instantiation of variables has a great chance to provide an improved error. The enumeration goes through two steps. A variable is first assigned with a floating-point value chosen randomly in its domain of values. Then, another random value chosen within the domain of its associated error is assign to the error associated to that variable. We exploit the fact that if the derivative sign does not change on the associated error domain, then the maximun distance between the hyperplan defined by the result over the floats and the related function over the reals is at one of the extrema of the associated error domain. When all variables from a function f representing a program have

Algorithm 1: branch-and-bound — maximization of error

Input : $\langle X, C, D\rangle$ — triple of variables, constraints, and domains
$e \in [-\infty, +\infty]$ — error to maximize
Output : $(e^*, \overline{e}, S)$
Data : $L \leftarrow \{ \prod_{x \in X} x \mid x = (\mathbf{x}, \mathbf{e_x})\}$ — set of boxes
$\overline{e} \leftarrow +\infty$ — dual bound
$e^* \leftarrow -\infty$ — primal bound
$\overline{e}^S \leftarrow -\infty$ — upper bound of sidelined boxes
$S \leftarrow \varnothing$ — stack of solutions

1 **while** $L \neq \varnothing$ **and** $e^* < \overline{e}$ **do**
  `/* Box selection: select a box` B `in the set of boxes` L `*/`
2 select $B \in L$; $L \leftarrow L \setminus B$
3 $\overline{e}^B_{old} \leftarrow \overline{e}^B$
4 $B \leftarrow \Phi(X, C \wedge e > e^*, B)$
5 **if** $\overline{e}^B_{old} = \overline{e}$ **and** ($B = \varnothing$ **or** $\overline{e}^B < \overline{e}$) **then**
6 **if** $B \neq \varnothing$ **then**
7 $\overline{e} \leftarrow \overline{e}^B$
8 **else**
9 $\overline{e} \leftarrow -\infty$
10 $\overline{e} \leftarrow \max\left(\{\overline{e}^{B_i} \mid \forall B_i \in L\} \cup \{\overline{e}, \overline{e}^S\}\right)$
11 **if** $B \neq \varnothing$ **then**
12 **if** $\overline{e}^B > e^*$ **then**
13 **if** $isBound(B)$ **then**
14 $(e^B, sol^B) \leftarrow (e^B, B)$
15 **else**
16 $(e^B, sol^B) \leftarrow primalComputation(B, X)$
17 **if** $e^B > e^*$ **then**
18 $e^* \leftarrow e^B$
19 push (e^B, sol^B) onto S
20 $L \leftarrow L \setminus \{B_i \in L \mid \overline{e}^{B_i} \leq e^*\}$
21 **if** $\overline{e}^B > \overline{e}^S$ **and** $\overline{e}^B > e^*$ **then**
  `/* Variable selection: select a variable` x `in box` B `*/`
22 **if** $(select(\mathbf{x}^B, \mathbf{e_x}^B) \in B \mid \underline{\mathbf{x}}^B < \overline{\mathbf{x}}^B)$ **and** $\neg isSidelined(B)$ **then**
  `/* Domain splitting: split the domain of values of x in subdomains */`
23 $B_1 \leftarrow B$
24 $B_2 \leftarrow B$
25 $\mathbf{x}^{B_1} \leftarrow \left[\underline{\mathbf{x}}^B, \frac{\underline{\mathbf{x}}^B + \overline{\mathbf{x}}^B}{2}\right]$
26 $\mathbf{x}^{B_2} \leftarrow \left[\left(\frac{\underline{\mathbf{x}}^B + \overline{\mathbf{x}}^B}{2}\right)^+, \overline{\mathbf{x}}^B\right]$
27 $L \leftarrow L \cup \{B_1, B_2\}$
28 **else**
29 $\overline{e}^S \leftarrow \max(\overline{e}^S, \overline{e}^B)$
30 **return** $(e^*, \overline{e}, S)$

been instantiated, an evaluation of f is done exactly over $\mathbb{Q}$ and in machine precision over $\mathbb{F}$. The error is exactly computed by evaluating $f_{\mathbb{Q}} - f_{\mathbb{F}}$ over $\mathbb{Q}$. The computed error can be further improved by a local search. That is to say, by exploring floating-point numbers around the chosen input values and evaluating again the expressions. This process is repeated a fixed number of times until the error can not be further improved, i.e., a local maximum has been reached. If the computed error is better than the primal, then e^* is updated. Each new primal bound is added to S alongside the input values exercising it.

Box splitting. A box is not split up but is discarded when its error upper bound is less than or equal to $\overline{e}^S$, the upper bound of sidelined boxes, or e^*. Discarding such a box speeds up solving time, since none of the errors contained in this box can improve the primal or the dual. Splitting occurs if and only if there exist at least one of the variables within B that is not instantiated and if the box is not sidelined. Otherwise, the box is sidelined and if $\overline{e}^B$ is strictly greater than $\overline{e}^S$, the latter is updated. The next variable to split on is selected in a lexicographic order. The bisection generates two sub-boxes that are added to L.

Note that Algorithm 1 always terminates and gives an enclosure of the maximal error: in the worst case, all boxes will be split up to degenerated boxes. Each degenerated box whose associated error e^B is lower than the primal will be discarded. If $e^* \leq e^B \leq \overline{e}$ holds, e^B will be used to update e^* and $\overline{e}$ before discarding the corresponding degenerated box. As a result, since the set of floating-point numbers is a finite set, the branch-and-bound requires a finite number of iterations to explore completely the initial box and thus, terminates.

5 Related work

Different tools exist for computing an over-approximation of errors of floating-point computations. Fluctuat [11, 10], is an abstract interpreter which combines affine arithmetic and zonotopes to analyze the robustness of programs over floats. FPTaylor [24, 23] uses symbolic Taylor expansions and global optimization to compute tight bounds of the error. It represents errors in Taylor series by a first order and a second order term. A branch-and-bound algorithm is used to compute an approximation of the symbolic first order error term, and the second order error term is computed in Taylor expansions. This branch-and-bound is very different from the one used by FErA. First, it considers only one term of the error representation used by FPTaylor whereas FErA branch-and-bound uses the whole error. Second, it does not compute a lower bound on the largest absolute error but only over-approximate errors. FPTaylor is also able to produce proof certificates to verify the validity of its computed bounds. Such a certificate can be externally checked in HOL Light [12]. PRECiSA [21, 27] is a static analyzer that also computes a certificate of proof that can be used to formally prove the round-off error bounds of a program. PRECiSA uses a branch-and-bound algorithm to compute concrete bounds of round-off errors. In other words, PRECiSA

computes symbolic error expressions to represent round-off errors that are given to a branch-and-bound algorithm. Certificate of proof produced by PRECiSA are validated by the PVS theorem prover [22]. Gappa [8] verifies properties on floating-point programs, and in particular computes bounds on round-off errors. Gappa works with an interval representation of floating-point numbers and applies rewriting rules for improving computed results. It is also able to generate formal proof of verified properties, that can in turn be checked in Coq [26]. Real2Float [17] uses semidefinite programming for estimating bounds on error. It decomposes an error into an affine part with respect to the error variable and a higher-order part. Bounds on the higher-order part are computed in the same way as FPTaylor. For the affine part, a relaxation procedure based on semidefinite programming is employed. Rosa [6, 7] uses affine arithmetic and an SMT solver to estimates round-off errors. It computes errors in a symbolic form that is given to the SMT solver to find concrete bounds on expressions. Daisy [15, 4, 5] is another tool by the authors of Rosa. It also relies on affine arithmetic and an SMT solver to bound round-off errors. Moreover, it also include features from FPTaylor, such as optimization-based absolute error analysis, and Fluctuat, such as interval subdivision. Most of these tools handle both transcendental functions and arithmetic operators. As said before, FErA only handles basic arithmetic operators (namely $+, -, \times, /$) because we want to provide a correct lower and upper bounds of the errors.

FPSDP [16] is a tool based on semidefinite programming that only computes under-approximation of largest absolute errors. In contrast to our approach FPSDP computes an under-approximation of the maximal error. The point is that this under-approximation might not be reachable. For instance, considers the benchmark `rigidBody1` (see Table 2). Here, FPSDP yields $3.55e{-}13$ whereas FErA, Gappa, Daisy, and FPTaylor upper bound is $2.95e{-}13$. Fluctuat and Rosa compute an upper bound of $3.22e{-}13$ and PRECiSA an upper bound of $3.23e{-}13$. The only upper bound greater than $3.55e{-}13$ is the one computed by Real2Float, about $5.33e{-}13$. Thus, the under-approximation of the maximal error provided by FPSDP is not reachable. S3FP [2] relies on random sampling and shadow values executions to find input values maximizing an error. It computes the error as the difference between the execution of a program done in a higher precision, acting as $\mathbb{R}$, and a lower precision, acting as $\mathbb{F}$. S3FP starts with an initial configuration that is cut into smaller configurations. Then, it selects a configuration and randomly instantiates variables to evaluate the program in both precisions. This process is repeated a finite number of time to improve the lower bound. Depending on the size of input intervals, S3FP can get stuck on a local maximum. To avoid this problem it uses a standard restart process. S3FP is the closest to our primal computation procedure. Both rely on random generation of input values to compute a lower bound of errors. However, as S3FP does all computations over $\mathbb{F}$, the resulting error suffers from rounding issues and thus, mights underestimate or overestimate the actual error. Such a computed error is unreachable. Furthermore, S3FP is highly reliant on the parametrized partitioning of the initial configuration. It cannot discard configurations where

	Fluctuat	Gappa	PRECiSA	Real2Float	Daisy	Rosa	FPTaylor	S3FP	FErA		
									filtering	e^*	$\overline{e}$
carbonGas	1.17e-08	*6.03e-09*	7.09e-09	2.21e-08	3.91e-08	1.60e-08	**4.96e-09**	4.2e-09	4.24e-08	2.95e-09	7.01e-09
	0.123s	3.445s	0.034s	6.887s	37.750s	37.581s	0.320s		0.017s		0.345s
verhulst	4.80e-16	*2.84e-16*	5.14e-16	4.66e-16	3.72e-16	4.67e-16	**2.47e-16**	2.4e-16	4.19e-16	2.19e-16	2.86e-16
	0.108s	0.619s	0.023s	4.675s	28.250s	15.762s	0.290s		0.016s		0.034s
predPrey	2.35e-16	*1.67e-16*	2.09e-16	2.51e-16	1.75e-16	1.98e-16	**1.59e-16**	1.5e-16	1.84e-16	1.03e-16	*1.67e-16*
	0.107s	2.166s	0.020s	7.269s	29.500s	33.220s	0.410s		0.011s		0.084s
rigidBody1	*3.22e-13*	**2.95e-13**	3.23e-13	5.33e-13	**2.95e-13**	*3.22e-13*	**2.95e-13**	2.7e-13	2.95e-13	1.95e-13	**2.95e-13**
	2.794s	2.359s	0.033s	3.230s	27.983s	7.505s	0.280s		0.018s		1.659s
rigidBody2	*3.65e-11*	**3.61e-11**	*3.65e-11*	6.48e-11	**3.61e-11**	*3.65e-11*	**3.61e-11**	3.0e-11	3.61e-11	2.52e-11	**3.61e-11**
	5.090s	3.657s	0.370s	3.698s	32.683s	10.377s	0.310s		0.022s		3.298s
doppler1	*1.27e-13*	1.61e-13	2.09e-13	7.64e-12	4.19e-13	2.68e-13	**1.22e-13**	1.0e-13	4.96e-13	7.34e-14	1.56e-13
	8.347s	5.542s	0.044s	26.821s	30.817s	24.298s	1.450s		0.021s		0.752s
doppler2	*2.35e-13*	2.86e-13	3.07e-13	8.85e-12	1.05e-12	6.45e-13	**2.23e-13**	1.9e-13	1.33e-12	1.12e-13	3.36e-13
	8.244s	5.634s	0.041s	26.731s	34.000s	24.073s	1.730s		0.034s		0.356s
doppler3	*7.12e-14*	8.75e-14	9.50e-14	4.07e-12	1.68e-13	1.01e-13	**6.62e-14**	5.7e-14	1.92e-13	4.09e-14	9.00e-14
	9.028s	5.476s	0.044s	26.057s	32.250s	31.442s	1.330s		0.023s		0.341s
turbine1	3.09e-14	2.41e-14	2.52e-14	2.46e-11	8.65e-14	5.99e-14	**1.67e-14**	1.1e-14	2.16e-13	1.05e-14	*1.76e-14*
	7.555s	9.816s	0.144s	127.911s	32.950s	31.400s	0.450s		0.016s		8.514s
turbine2	2.59e-14	3.32e-14	3.01e-14	2.07e-12	1.31e-13	7.67e-14	**2.00e-14**	1.4e-14	3.04e-13	1.32e-14	*2.36e-14*
	5.562s	7.395s	0.132s	22.225s	30.183s	14.890s	0.560s		0.025s		2.803s
turbine3	1.34e-14	0.35	1.83e-14	1.70e-11	6.23e-14	4.62e-14	**9.57e-15**	6.2e-15	1.56e-13	4.76e-15	*1.10e-14*
	7.342s	11.256s	0.193s	150.653s	31.050s	31.224s	0.520s		0.026s		2.766s
sqroot	6.83e-16	5.35e-16	**4.29e-16**	1.28e-15	5.71e-16	6.18e-16	*5.02e-16*	4.7e-16	5.78e-16	3.33e-16	5.33e-16
	0.120s	7.937s	0.035s	13.840s	28.000s	8.414s	0.320s		0.032s		2.989s
sine	7.41e-16	6.95e-16	7.48e-16	6.03e-16	1.13e-15	*5.18e-16*	**4.44e-16**	2.9e-16	7.41e-16	2.24e-16	7.41e-16
	0.126s	40.351s	0.132s	13.138s	27.933s	14.265s	0.450s		0.027s		12.927s
sineOrder3	1.09e-15	6.54e-16	1.23e-15	1.19e-15	1.45e-15	9.96e-16	**5.94e-16**	4.1e-16	1.11e-15	3.28e-16	*6.36e-16*
	0.117s	3.177s	0.021s	4.241s	25.867s	6.974s	0.290s		0.021s		1.388s
kepler0	1.03e-13	1.09e-13	1.10e-13	1.20e-13	1.04e-13	*8.28e-14*	**7.47e-14**	5.3e-14	1.18e-13	5.43e-14	9.81e-14
	12.611s	12.187s	0.230s	2.120s	28.033s	11.113s	0.690s		0.037s		TO
kepler1	3.51e-13	4.68e-13	4.03e-13	4.67e-13	4.81e-13	4.14e-13	**2.86e-13**	1.6e-13	4.94e-13	1.41e-13	*3.10e-13*
	252.468s	19.785s	0.683s	93.202s	28.933s	134.149s	1.710s		0.031s		51.303s
kepler2	2.24e-12	2.40e-12	*1.66e-12*	2.09e-12	2.46e-12	2.15e-12	**1.58e-12**	8.4e-13	2.43e-12	6.08e-13	1.83e-12
	33.600s	39.048s	31.235s	59.881s	30.483s	72.847s	0.580s		0.027s		157.558s

Table 2. Experimental results for absolute round-off error bounds (bold indicates the **best approximation**, italic indicates the *second best*, and red indicates the worst one). Grey rows indicate solving time for each tool, in seconds. S3FP and e^* columns show lower bound on the maximal error, whereas other columns show an upper bound. TO indicates a time out at 10 minutes

no improvement of the lower bound is possible. In contrast, FErA selects boxes to explore on the basis of their upper bounds to try finding a better lower bound.

6 Experimentation

In this section, we provide preliminary experiments of FErA on a subset of benchmarks (comprising of $+, -, \times, /$ operators) from the FPBench [3] suite, a common standard to compare verification tools over floating-point numbers. Table 2 compares results from Fluctuat [11, 10] (version 3.1390 with subdivisions), Gappa [8] (version 1.3.5 with advanced hints), PRECiSA [21, 27] (version 2.1.1), Real2Float [17] (version 0.7), Daisy [15, 4, 5] (master branch, commit 8f26766), Rosa [6, 7] (master branch, commit 68e58b8), FPTaylor [24, 23] (master branch, commit 147e1fe with Gelpia [23] optimizer), S3FP [2] and FErA. Benchmarks

computation is done on a 2.8 GHz Intel Core i7–7700HQ with 16 GB of RAM, running under macOs Catalina (10.15.4). Results from S3FP are taken from [23], as authors state in [2] that the available tool only works on single-precision floating-point numbers.

Note that all state-of-the-art tools provide an over-approximation of errors, except S3FP, which compute a lower bound on largest absolute errors. For FErA, column *filtering* gives the over-approximation computed by a single filtering while column e^* and column $\overline{e}$ provide respectively the best reachable error and over-approximation of the error computed by FErA. Bold and italic are used to rank, respectively, the best and second best over-approximation while red indicates the worst ones. Lines in grey give the time in second to compute these bounds.

On these benchmarks, FErA classified as best twice and as second six times. Note that it never provides the worst result. In almost all cases, the computed reachable error e^* is in the same order of magnitude than $\overline{e}$. The lack of dedicated handling of multiple occurrences in FErA is underlined by the computed dual of the `sine` bench. Here, the splitting process used in the branch-and-bound is not sufficient to lower the dual value. FErA solves most of the problems in a reasonable amount of time with the exception of `kepler0`. Indeed, Kepler benchs are the problems with the biggest number of input variables and FErA performs better on small sized problems. Still, FErA as an anytime algorithm provides bounds computed so far for `kepler0`.

7 Conclusion

This paper addresses a critical issue in program verification: computing an enclosure of the maximal absolute error in floating-point computations. To compute this enclosure, we introduce an original approach based on a branch-and-bound algorithm using the constraint system for round-off error analysis from [9].

The splitting process takes advantage of an efficient filtering and the known enclosure of the error to speed up the optimization process.

Alongside a rigorous enclosure of maximal errors, the algorithm provides input values exercising the lower bound. Knowing such bounds of the maximal error is very useful to get rid of false positives, a critical issue in program verification and validation.

Preliminary experiments on a set of standard benchmarks are very promising and compare well to other available tools.

Further works include a better understanding and a tighter computation of round-off errors to smooth the effects of the dependency problem, experimentations with different search strategies dedicated to floating-point numbers [29] to improve the resolution process, as well as devising a better local search to speed up the primal computation procedure.

References

1. Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. Software Testing, Verification and Reliability **16**(2), 97–121 (2006)
2. Chiang, W., Gopalakrishnan, G., Rakamaric, Z., Solovyev, A.: Efficient search for inputs causing high floating-point errors. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014. pp. 43–52 (2014)
3. Damouche, N., Martel, M., Panchekha, P., Qiu, C., Sanchez-Stern, A., Tatlock, Z.: Toward a standard benchmark format and suite for floating-point analysis. In: 9th International Workshop on Numerical Software Verification (NSV2017). pp. 63–77 (2017)
4. Darulova, E., Horn, E., Sharma, S.: Sound mixed-precision optimization with rewriting. In: Gill, C., Sinopoli, B., Liu, X., Tabuada, P. (eds.) Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018, Porto, Portugal, April 11-13, 2018. pp. 208–219. IEEE Computer Society / ACM (2018)
5. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - framework for analysis and optimization of numerical programs (tool paper). In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10805, pp. 270–287. Springer (2018)
6. Darulova, E., Kuncak, V.: Sound compilation of reals. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 235–248. ACM (2014)
7. Darulova, E., Kuncak, V.: Towards a compiler for reals. ACM Trans. Program. Lang. Syst. **39**(2), 8:1–8:28 (2017)
8. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. ACM Trans. Math. Softw. **37**(1), 2:1–2:20 (2010)
9. Garcia, R., Michel, C., Pelleau, M., Rueher, M.: Towards a constraint system for round-off error analysis of floating-point computation. In: 24th International Conference on Principles and Practice of Constraint Programming :Doctoral Program. Lille, France (Aug 2018)
10. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4134, pp. 18–34 (2006)
11. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011). pp. 232–247 (2011)
12. Harrison, J.: HOL Light: An overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009. Lecture Notes in Computer Science, vol. 5674, pp. 60–66. Springer-Verlag, Munich, Germany (2009)
13. Hauser, J.R.: Handling floating-point exceptions in numeric programs. ACM Trans. Program. Lang. Syst. **18**(2), 139–174 (Mar 1996)
14. IEEE: 754-2008 - IEEE Standard for floating point arithmethic (2008)
15. Izycheva, A., Darulova, E.: On sound relative error bounds for floating-point arithmetic. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer

Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017. pp. 15–22. IEEE (2017)
16. Magron, V.: Interval Enclosures of Upper Bounds of Roundoff Errors Using Semidefinite Programming. ACM Trans. Math. Softw. **44**(4), 41:1–41:18 (2018)
17. Magron, V., Constantinides, G.A., Donaldson, A.F.: Certified roundoff error bounds using semidefinite programming. ACM Trans. Math. Softw. **43**(4), 34:1–34:31 (2017)
18. Marre, B., Michel, C.: Improving the floating point addition and subtraction constraints. In: Proceedings of the 16th international conference on Principles and practice of constraint programming (CP'10). pp. 360–367. LNCS 6308, St. Andrews, Scotland (6–10th Sep 2010)
19. Michel, C.: Exact projection functions for floating point number constraints. In: AI&M 1-2002, Seventh international symposium on Artificial Intelligence and Mathematics (7th ISAIM). Fort Lauderdale, Floride (US) (2–4th Jan 2002)
20. Michel, C., Rueher, M., Lebbah, Y.: Solving constraints over floating-point numbers. In: 7th International Conference on Principles and Practice of Constraint Programming (CP 2001). pp. 524–538 (2001)
21. Moscato, M., Titolo, L., Dutle, A., Muñoz, C.A.: Automatic estimation of verified floating-point round-off errors via static analysis. In: Computer Safety, Reliability, and Security. pp. 213–229 (2017)
22. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings. Lecture Notes in Computer Science, vol. 607, pp. 748–752. Springer (1992)
23. Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. ACM Trans. Program. Lang. Syst. **41**(1), 2:1–2:39 (Dec 2018)
24. Solovyev, A., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In: Bjørner, N., de Boer, F. (eds.) FM 2015: Formal Methods. pp. 532–550. Springer International Publishing, Cham (2015)
25. Sterbenz, P.H.: Floating Point Computation. Prentice-Hall (1974)
26. The Coq Development Team: The Coq proof assistant reference manual (2020), https://coq.inria.fr, version 8.11.2
27. Titolo, L., Feliú, M.A., Moscato, M.M., Muñoz, C.A.: An abstract interpretation framework for the round-off error analysis of floating-point programs. In: Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9. pp. 516–537 (2018)
28. Titolo, L., Muñoz, C.A., Feliú, M.A., Moscato, M.M.: Eliminating unstable tests in floating-point programs. In: Mesnard, F., Stuckey, P.J. (eds.) Logic-Based Program Synthesis and Transformation - 28th International Symposium, LOPSTR 2018, Frankfurt/Main, Germany, September 4-6, 2018, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11408, pp. 169–183. Springer (2018)
29. Zitoun, H.: Search strategies for solving constraint systems over floats for program verification. Theses, Université Côte d'Azur (Oct 2018)
30. Zitoun, H., Michel, C., Rueher, M., Michel, L.: Search strategies for floating point constraint systems. In: 23rd International Conference on Principles and Practice of Constraint Programming, CP 2017. pp. 707–722 (2017)