A branch-and-bound algorithm to rigorously enclose the round-off errors*

Rémy Garcia, Claude Michel, and Michel Rueher

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

Abstract. Round-off errors occur each time a program makes use of floating-point computations. They denote the existence of a distance between the actual floating-point computations and the intended computations over the reals. Therefore, analyzing round-off errors is a key issue in the verification of programs with floating-point computations. Most existing tools for round-off error analysis provide an over-approximation of the error. The point is that these approximations are often too coarse to evaluate the effective consequences of the error on the behaviour of a program. Some other 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 branch-and-bound algorithm to rigorously enclose the maximal error. Thanks to the use of rational arithmetic, our branch-and-bound algorithm provides a tight upper bound of the maximal error and a lower bound that can be exercised by input values. We outline the advantages and limits of our framework and compare it with state-of-the-art methods. Preliminary experiments on standard benchmarks show promising results.

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 faithful 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 [7, 6] is an abstract interpreter that combines affine arithmetic and zonotopes to analyze the robustness of programs over the floats, FPTaylor [19, 18] uses symbolic Taylor expansions to compute tight bounds of the error, and PRECiSA [16, 22] 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 [11] which takes advantage of 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 here.
The core of the proposed approach is a branch-and-bound algorithm that attempts to maximize a given error of a program with floating-point computations. This branch-and-bound algorithm is embedded in a solver over the floats [24, 13, 1, 14, 15] extended to constraints over errors [5]. 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. 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 rely on each other for improvement.
Maximizing an error can be very expensive for the errors that 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, and thus it always provides an enclosure of the maximal error alongside input values exercising the lower bound.

1.1 Motivating example

Consider the piece of code in Example 1 that computes $z = (3 * x + y)/w$ using 64 bits doubles 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. When z is lower than 10, with a tolerance to errors of δ, values supported by z are considered as safe and related computations can be done. Otherwise, an alarm must be raised.

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

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

Example 1. Simple program

Tool	Error
FPTaylor	5.15e-15
PRECiSA	5.08e-15
Fluctuat	6.28e-15
FErA	4.96e-15

Table 1. Absolute error bound

Now, assume that δ is set to $5.0e{-}15$. 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 $\mathbb{R}$. Table 1 reports the error values given by FPTaylor [19, 18], PRECiSA [22], Fluctuat [7, 6], and our tool FErA. All analyzers but FErA compute a bound greater than δ. Results from FPTaylor, PRECiSA and Fluctuat suggests that the alarm might inappropriately be raised.
FErA computes a round-off error 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:

$$\begin{aligned} x &= 8.9999999999996624922 & e_x &= -8.8817841970012523239e{-}16 \\ y &= 4.9999999999994848565 & e_y &= -4.4408920985006261616 9e{-}16 \\ w &= 3.1999999999998419042 & e_w &= +2.2204460492503130808 5e{-}16 \\ z &= 10.0000000000000035527 & e_z &= -3.5527136788005009293 6e{-}15 \end{aligned}$$

In other words, our sound optimizer not only guarantees that, despite errors in floating-point computations, this program can never raise an alarm when $z <= 10$ over the reals, but it also provides an enclosure of the largest absolute error. Such an enclosure having a ratio[1] of ≈ 1.4 shows that the round-off error bounds of FErA are close to the actual error. Note that the computed lower bound corresponds to a case where z over the floats is bigger than 10 while it remains lower than 10 over the reals.
Now, assume that δ is set to $3.00e{-}15$. The issue here is to know whether there exists at least one case where raiseAlarm() is reached when z is less than or equal to 10 on real numbers. The previously computed enclosure of the maximal error provided by FErA ensures that there exist at least one case where raiseAlarm() is reached with an error bigger than δ. The other tools 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

[1] The ratio between FErA computed upper and lower bound is equal to $4.96e{-}15/3.5527e{-}15 = 1.396$.

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 on computing a lower bound on the maximal error and their pitfalls. Section 6 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, i.e. $+, -, \times$ and $/$, for which the error can be computed exactly using rational arithmetic [5]. 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 rational numbers is denoted $\mathbb{Q}$, and the set of real numbers is denoted $\mathbb{R}$. $\mathbb{F}$ denotes a set of floating-point numbers whose precision is one of the precision defined in IEEE 754 [10]. Though the approach presented here applies to other types of floats, in the rest of the paper, $\mathbb{F}$ will denote 64 bits floating-point numbers unless otherwise stated. For each floating-point variable x in X, the domain of values of x is represented by the interval $\mathbf{x} = [\underline{\mathbf{x}}, \overline{\mathbf{x}}] = \{x \in \mathbb{F},\ \underline{\mathbf{x}} \leq x \leq \overline{\mathbf{x}}\}$, where $\underline{\mathbf{x}} \in \mathbb{F}$ and $\overline{\mathbf{x}} \in \mathbb{F}$. $\underline{\mathbf{x}}$ (resp. $\overline{\mathbf{x}}$) denotes the lower (resp. upper) bound of the interval $\mathbf{x}$. The domain of errors $\mathbf{e_x}$ of x is represented by an interval of rationals $\mathbf{e_x} = [\underline{\mathbf{e}}_\mathbf{x}, \overline{\mathbf{e}}_\mathbf{x}] = \{e_x \in \mathbb{Q},\ \underline{\mathbf{e}}_\mathbf{x} \leq e_x \leq \overline{\mathbf{e}}_\mathbf{x}\}$ where $\underline{\mathbf{e}}_\mathbf{x} \in \mathbb{Q}$ and $\overline{\mathbf{e}}_\mathbf{x} \in \mathbb{Q}$. $x_\mathbb{F}$ (respectively, $x_\mathbb{Q}$ and $x_\mathbb{R}$) 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, or a singleton, i.e. when $\underline{\mathbf{e}}_\mathbf{x} = \overline{\mathbf{e}}_\mathbf{x}$.
The branch-and-bound algorithm maximizes an error noted e that results from floating-point computations along a given path in a program. e^* denotes the lower bound, i.e. the maximal error computed so far while $\overline{e}$ denotes the upper bound, i.e. the currently best known over-approximation of the error. Both of those bounds are expressed in absolute value. S is the ordered, by error values, 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.

3 A constraint system for round-off error

The branch-and-bound algorithm at the core of our framework is based on a constraint system on errors [5] that we briefly describe in this section.

3.1 Computing rounding errors

The IEEE 754 standard [10] requires a correct rounding for the four basic operations of the floating-point arithmetic. The result of such an operation over the

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

Fig. 1. Computation of deviation for basic operations

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 $\pm\frac{1}{2}ulp(z)$[2] 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: $e_z = e_x - e_y + ((x_\mathbb{F} - y_\mathbb{F}) - (x_\mathbb{F} \ominus y_\mathbb{F}))$, where $(x_\mathbb{F} - y_\mathbb{F}) - (x_\mathbb{F} \ominus y_\mathbb{F})$ characterizes the error produced by the subtraction operation itself. Let's $e_\ominus$ denotes the error produced by the subtraction operation. The formula can then be denoted by: $e_z = e_x - e_y + e_\ominus$, that combines deviations from input values and the deviation introduced by the elementary operation.

Computation of 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.

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

3.2 A constraint network with three domains

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

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

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:

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

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 of standard floating-point projection functions defined in [14] and extended in [1, 13]. However, dedicated projections are required to reduce domains of errors.
The projections on the domains of errors are obtained using the natural extension over intervals of the formulae of Figure 1. Since these are formulae on reals, they can naturally be extended to intervals. The projections functions for the four basic arithmetic operations are detailed in Figure 2.
As 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.

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 x^{-} and x^{+} denote respectively, the greatest floating-point number strictly inferior to x and the smallest floating-point number strictly superior to x. This

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

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 [20] or the Hauser property on the addition [9]. Both properties give conditions under which these operations produce exact results, the same being true for the well-known prop-

erty that states that $2^k \otimes x$ is exactly computed, provided that no overflow occurs.

3.5 A CSP over $\mathbb{F}$ with errors

A CSP over $\mathbb{F}$ with errors is made of constraints over $\mathbb{F}$ with variables whose domains specify the allowed values over $\mathbb{F}$, as well as, the allowed values over $\mathbb{Q}$ of the associated error. $err(x)$, which denotes the error associated to variable x, permits constraints on errors. Note that the latter are constraints over $\mathbb{Q}$.
The constraint network results from elementary constraints issued from the decomposition of initial constraints. Each elementary constraint is in charge of applying the set of related projection functions to reduce the domain of the variables involved in the constraint. They compute a quasi-fixpoint, set to 5%, in order to avoid some slow convergence issues. Propagation occurs following an AC3 algorithm.

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

Our branch-and-bound algorithm (see Algorithm 1) maximizes 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 or minimize a signed error.
The branch-and-bound algorithm takes as inputs a CSP $\langle X, C, D\rangle$, and e, an error to maximize. It computes a lower bound, e^* and an upper bound, $\overline{e}$ of the maximal error e. e^* and $\overline{e}$ bounds the maximal error: $e^* \leq e \leq \overline{e}$. The computed lower bound e^* is a reachable error exercised by computed input values. These values and the computed bounds are returned by the algorithm.

Stopping criteria. The primary aim of the branch-and-bound algorithm is to compute the maximal error. This is achieved when the lower bound is equal to the upper bound. However, such a condition may be difficult to meet.
A first issue comes from the dependency problem which appears on expressions with multiple 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 arises in projection functions of errors that contain multiple occurrences like in multiplication and division. It can lead to unnecessary over-approximations of resulting intervals. A direct consequence of this problem is that the upper bound 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. 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 lower and upper bounds 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 upper bound. Solving stops when there are no more boxes to process or when the lower bound e^* and the upper bound $\overline{e}$ 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, upper bound updating, lower bound 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 a constraint on the known lower bound of e, i.e. $e^* \leq e$. If $\Phi(B) = \varnothing$, the selected box does not contain any solution; either because it contradicts one of the initial constraints or because of known bounds of constraints over e. In both cases, the algorithm discards box B and directly jumps to the next loop iteration.

Upper bound update. Once B has been filtered, if the error upper bound of the current box was the support of the global $\overline{e}$ and is no longer, then $\overline{e}$ is updated.

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
$\overline{e} \leftarrow +\infty$ — upper bound
$e^* \leftarrow -\infty$ — lower 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 $B \in L$; $L \leftarrow L \setminus B$; $\overline{e}^B_{old} \leftarrow \overline{e}^B$
3 $B \leftarrow \Phi(X, C \wedge e > e^*, B)$
4 **if** $\overline{e}^B_{old} = \overline{e}$ **and** ($B = \varnothing$ **or** $\overline{e}^B < \overline{e}$) **then**
5 **if** $B \neq \varnothing$ **then** $\overline{e} \leftarrow \overline{e}^B$ **else** $\overline{e} \leftarrow -\infty$
6 $\overline{e} \leftarrow \max\left(\{\overline{e}^{B_i} \mid \forall B_i \in L\} \cup \{\overline{e}, \overline{e}^S\}\right)$
7 **if** $B \neq \varnothing$ **then**
8 **if** $\overline{e}^B > e^*$ **then**
9 **if** $isBound(B)$ **then**
10 $(e^B, sol^B) \leftarrow (e^B, B)$
11 **else**
12 $(e^B, sol^B) \leftarrow LowerBounding(B, X)$
13 **if** $e^B > e^*$ **then**
14 $e^* \leftarrow e^B$; push (e^B, sol^B) onto S
15 $L \leftarrow L \setminus \{B_i \in L \mid \overline{e}^{B_i} \leq e^*\}$
16 **if** $\overline{e}^B > \overline{e}^S$ **and** $\overline{e}^B > e^*$ **then**
  `/* Variable selection: select a variable` x `in box` B `*/`
17 **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: on the domain of values of x */`
18 $B_1 \leftarrow B$; $B_2 \leftarrow B$
19 $\mathbf{x}^{B_1} \leftarrow \left[\underline{\mathbf{x}}^B, \frac{\underline{\mathbf{x}}^B + \overline{\mathbf{x}}^B}{2}\right]$; $\mathbf{x}^{B_2} \leftarrow \left[\left(\frac{\underline{\mathbf{x}}^B + \overline{\mathbf{x}}^B}{2}\right)^+, \overline{\mathbf{x}}^B\right]$
20 $L \leftarrow L \cup \{B_1, B_2\}$
21 **else** $\overline{e}^S \leftarrow \max(\overline{e}^S, \overline{e}^B)$
22 **return** $(e^*, \overline{e}, S)$

$\overline{e}$ is updated with the maximum among the upper bound of errors of the current box, of remaining boxes in L, and of sidelined boxes.

Lower bound update. A non empty box may contain a better lower bound than the current one. A *generate-and-test* procedure, `LowerBounding`, attempts to compute a better one through a two-step process. 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 assigned 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 maximum 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. In such a case, the random instantiation of the error is restricted to the corners of its domain. When all variables from a function f representing a program have 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 current lower bound, then e^* is updated. Each new reachable lower 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 lower or the upper bounds. 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 round-robin on 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 current lower bound 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 to compute an over-approximation of floating-point computation round-off errors. Fluctuat [7, 6], is an abstract interpreter that combines affine arithmetic and zonotopes to analyze the robustness of programs over the floats. FPTaylor [19, 18] relies on symbolic Taylor expansions and global optimization to compute tight bounds of the error. It makes use of Taylor series of first and second order to evaluate the error. A branch-and-bound algorithm approximates the first order error terms while the second order error terms are directly bounded by means of interval arithmetic. This branch-and-bound is very

different from the one of FErA. First, it considers only the first order terms of the error whereas FErA branch-and-bound works on the whole error. Second, it does not compute a lower bound on the largest absolute error but only over-approximate the first order terms of the error. FPTaylor also generates a proof certificate of its computed bounds. Such a certificate can be externally checked in HOL Light [8]. The static analyzer PRECiSA [16, 22] computes also a certificate of proof that can be validated by the PVS theorem prover [17]. PRECiSA computes symbolic error expressions to represent round-off errors that are then given to a branch-and-bound algorithm to get the error bounds. Gappa [4] 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 [21]. Real2Float [12] uses semidefinite programming to estimate bounds on errors. 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.
FPSDP [11] 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. S3FP [2] relies on random sampling and shadow value 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 precision. This process is repeated a finite number of time to improve the lower bound. Depending on the size of input variable domains, S3FP can get stuck on a local maximum. To avoid this problem it uses a standard restart process. S3FP is the closest to our lower bound 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, might 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 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 Experiments

In this section, we provide preliminary experiments of FErA on benchmarks from the FPBench [3] suite, a common standard to compare verification tools over floating-point numbers. Table 2 compares the behaviour of FErA with different

	filtering	$e^* = \overline{e}$		$e^* = \overline{e}$ w. s.		$\frac{\overline{e}}{e^*} \leq 2$		$\frac{\overline{e}}{e^*} \leq 2$ w. s.	
		e^*	$\overline{e}$	e^*	$\overline{e}$	e^*	$\overline{e}$	e^*	$\overline{e}$
carbonGas	4.2e-8	4.2e-9	6.0e-9	2.9e-9	7.0e-9	3.6e-9	7.0e-9	3.6e-9	7.0e-9
	0.017s		TO		0.345s		1.419s		0.266s
verhulst	4.2e-16	2.4e-16	2.8e-16	2.1e-16	2.9e-16	1.8e-16	2.9e-16	1.6e-16	3.0e-16
	0.016s		TO		0.034s		0.024s		0.018s
predPrey	1.8e-16	1.5e-16	1.7e-16	1.0e-16	1.7e-16	9.3e-17	1.7e-16	9.8e-17	1.8e-16
	0.011s		TO		0.084s		0.041s		0.018s
rigidBody1	2.9e-13	2.8e-13	2.9e-13	1.9e-13	2.9e-13	1.4e-13	2.9e-13	1.4e-13	2.9e-13
	0.018s		TO		1.659s		0.370s		0.543s
rigidBody2	3.6e-11	3.1e-11	3.6e-11	2.5e-11	3.6e-11	1.8e-11	3.6e-11	2.1e-11	3.6e-11
	0.022s		TO		3.298s		1.367s		1.266s
doppler1	5.0e-13	1.1e-13	1.5e-13	7.3e-14	1.6e-13	1.0e-13	1.5e-13	7.7e-14	1.5e-13
	0.021s		TO		0.752s		1.099s		0.757s
doppler2	1.3e-12	2.1e-13	2.7e-13	1.1e-13	3.4e-13	1.3e-13	2.7e-13	1.0e-13	3.4e-13
	0.034s		TO		0.356s		1.416s		0.378s
doppler3	1.9e-13	6.2e-14	8.4e-14	4.0e-14	9.0e-14	4.4e-14	8.7e-14	3.9e-14	9.0e-14
	0.023s		TO		0.341s		0.455s		0.311s
turbine1	2.2e-13	1.3e-14	1.7e-14	1.0e-14	1.8e-14	1.0e-14	2.0e-14	9.3e-15	1.8e-14
	0.016s		TO		8.514s		2.289s		6.042s
turbine2	3.0e-14	1.5e-14	2.3e-14	1.3e-14	2.4e-14	1.3e-14	2.4e-14	1.1e-14	2.4e-14
	0.025s		TO		2.803s		1.581s		2.952s
turbine3	1.6e-13	6.4e-15	1.1e-14	4.7e-15	1.1e-14	5.7e-15	1.1e-14	5.6e-15	1.1e-14
	0.026s		TO		2.766s		3.961s		1.800s
sqroot	5.8e-16	4.5e-16	5.3e-16	3.3e-16	5.3e-16	2.9e-16	5.8e-16	3.3e-16	5.8e-16
	0.032s		TO		2.989s		0.277s		0.183s
sine	7.4e-16	2.8e-16	7.4e-16	2.2e-16	7.4e-16	2.6e-16	7.4e-16	2.4e-16	7.4e-16
	0.027s		TO		12.927s		TO		14.833s
sineOrder3	1.1e-15	4.0e-16	6.4e-16	3.2e-16	6.4e-16	3.1e-16	6.4e-16	3.3e-16	6.4e-16
	0.021s		TO		1.388s		1.433s		1.504s
kepler0	1.2e-13	5.7e-14	9.8e-14	5.4e-14	9.8e-14	5.0e-14	9.8e-14	4.9e-14	9.8e-14
	0.037s		TO		TO		1.937s		2.798s
kepler1	4.9e-13	1.6e-13	3.1e-13	1.4e-13	3.1e-13	1.6e-13	3.1e-13	1.6e-13	3.1e-13
	0.031s		TO		51.303s		15.136s		12.691s
kepler2	2.4e-12	7.9e-13	1.8e-12	6.5e-13	1.8e-12	6.9e-13	1.8e-12	6.7e-13	1.8e-12
	0.027s		TO		58.834s		TO		72.622s

Table 2. Comparison of stopping criteria.

stopping criteria while Table 3 compares results from FErA with state-of-the art tools, namely Gappa [4], Fluctuat [7, 6], Real2Float [12], FPTaylor [19, 18], PRECiSA [16, 22], and S3FP [2]. Results from all tools but FErA are taken from [18].

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,

	Fluctuat	Gappa	PRECiSA	Real2Float	FPTaylor	FErA		S3FP
						e^*	$\overline{e}$	
carbonGas	1.2e-8	*6.1e-9*	7.4e-9	2.3e-8	**6.0e-9**	3.6e-9	7.0e-9	4.2e-9
	0.54s	2.35s	6.30s	4.65s	1.08s		0.266s	
verhulst	4.9e-16	*2.9e-16*	*2.9e-16*	4.7e-16	**2.5e-16**	1.6e-16	3.0e-16	2.4e-16
	0.09s	0.41s	4.95s	2.52s	0.99s		0.018s	
predPrey	2.4e-16	*1.7e-16*	*1.7e-16*	2.6e-16	**1.6e-16**	9.8e-17	1.8e-16	1.5e-16
	0.18s	1.40s	8.08s	4s	1.07s		0.018s	
rigidBody1	3.3e-13	*3.0e-13*	*3.0e-13*	5.4e-13	*3.0e-13*	1.4e-13	**2.9e-13**	2.7e-13
	1.96s	1.42s	7.42s	3.09s	0.99s		0.543s	
rigidBody2	*3.7e-11*	*3.7e-11*	*3.7e-11*	6.5e-11	*3.7e-11*	2.1e-11	**3.6e-11**	3.0e-11
	3.87s	2.22s	10.79s	1.08s	1.02s		1.266s	
doppler1	**1.3e-13**	1.7e-13	2.7e-13	7.7e-12	**1.3e-13**	7.7e-14	*1.5e-13*	1.0e-13
	6.30s	3.31s	16.17s	13.20s	1.97s		0.757s	
doppler2	*2.4e-13*	2.9e-13	5.4e-13	1.6e-11	**2.3e-13**	1.0e-13	3.4e-13	1.9e-13
	6.15s	3.37s	16.87s	13.33s	2.20s		0.378s	
doppler3	*7.2e-14*	8.7e-14	1.4e-13	8.6e-12	**6.7e-14**	3.9e-14	9.0e-14	5.7e-14
	6.46s	3.32s	15.65s	13.05s	1.88s		0.311s	
turbine1	3.1e-14	2.5e-14	3.8e-14	2.5e-11	**1.7e-14**	9.3e-15	*1.8e-14*	1.1e-14
	5.05s	5.54s	24.35s	136.35s	1.10s		6.042s	
turbine2	2.6e-14	3.4e-14	3.1e-14	2.1e-12	**2.0e-14**	1.1e-14	*2.4e-14*	1.4e-14
	3.98s	3.94s	19.17s	8.30s	1.17s		2.952s	
turbine3	1.4e-14	0.36	2.3e-14	1.8e-11	**9.6e-15**	5.6e-15	*1.1e-14*	6.2e-15
	5.08s	6.29s	24.47s	137.36s	1.21s		1.800s	
sqroot	6.9e-16	*5.4e-16*	6.9e-16	1.3e-15	**5.1e-16**	3.3e-16	5.8e-16	4.7e-16
	0.09s	5.06s	8.18s	4.23s	1.02s		0.183s	
sine	7.5e-16	7.0e-16	*6.0e-16*	6.1e-16	**4.5e-16**	2.4e-16	7.4e-16	2.9e-16
	0.11s	25.43s	11.76s	4.95s	1.14s		14.833s	
sineOrder3	1.1e-15	6.6e-16	1.2e-15	1.2e-15	**6.0e-16**	3.3e-16	*6.4e-16*	4.1e-16
	0.09s	2.09s	6.11s	2.22s	1.02s		1.504s	
kepler0	1.1e-13	1.1e-13	1.2e-13	1.2e-13	**7.5e-14**	4.9e-14	*9.8e-14*	5.3e-14
	8.59s	7.33s	37.57s	0.76s	1.31s		2.798s	
kepler1	3.6e-13	4.7e-13	crash	4.7e-13	**2.9e-13**	1.6e-13	*3.1e-13*	1.6e-13
	157.74s	10.68s	N/A	22.53s	2.08s		12.691s	
kepler2	2.3e-12	2.4e-12	crash	2.1e-12	**1.6e-12**	6.7e-13	*1.8e-12*	8.4e-13
	22.41s	24.17s	N/A	16.53s	1.3s		72.622s	

Table 3. Comparison of FErA with other tools.

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. Lines in grey give the time in second to compute these bounds. Note that experiments have been made with a timeout, noted TO, of 10min. In Table 3, bold and italic are used to rank, respectively, the best and second best over-approximation while red indicates the worst ones.

Table 2 compares the behaviour of FErA with different stopping criteria. The ideal $e^* = \overline{e}$ criterion stops if and only if the lower bound reaches the upper bound. Of course, this criterion is hard to reach and benches are stopped by the timeout. But these columns yield among the best e^* and $\overline{e}$ that can be expected with FErA. For instance, most of the e^* provided here are better than the best known values[3], and the values of $\overline{e}$ are obviously the best values obtained by FErA. $e^* = \overline{e}$ w. s. combines the ideal criterion with sidelined boxes, i.e. boxes that are sidelined once all operations involved in the CSP produce an operation error, $e_{\odot}$, less or equal to half an ulp. Such a combination of criterion allows FErA to compute the results in a reasonable amount of time for all benches but one. However, this is obtained at the price of a degradation of the error bounds. $\frac{\overline{e}}{e^*} \leq 2$ relaxes the ideal stopping criterion to a ratio of 2. Here, two benchmarks reach the timeout. This is to be expected: the results with the ideal criterion show that FErA have difficulties to reach this ratio on two benchmarks, probably due to a huge amount of multiple occurrences. $\frac{\overline{e}}{e^*} \leq 2$ w. s. combines the ratio of 2 with sidelined boxes and avoid any timeout, though at the price of looser bounds.
As shown in Table 3, FErA classified as best twice and as second eight 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 as $\overline{e}$. The lack of dedicated handling of multiple occurrences in FErA is underlined by the computed upper bound of the `sine` bench. Here, the splitting process used in the branch-and-bound is not sufficient to lower the upper bound value. With the last combination of criteria, FErA solves most of the problems in a reasonable amount of time with the exception of `kepler2`. Indeed, Kepler benches are the problems with the biggest number of input variables and FErA performs better on small-sized problems.

7 Conclusion

This paper addresses a critical issue in program verification: computing an enclosure of the maximal absolute round-off errors that occur in floating-point computations. To do so, we introduce an original approach based on a branch-and-bound algorithm using the constraint system for round-off error analysis from [5]. Alongside a rigorous enclosure of maximal errors, the algorithm provides input values exercising the lower bound. Knowing such bounds of the maximal error allows 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, exploring different search strategies dedicated to floating-point numbers [23] to improve the resolution process, as well as devising a better local search to speed up the reachable lower bound computation procedure.

[3] See column S3FP in Table 3.

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. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. ACM Trans. Math. Softw. **37**(1), 2:1–2:20 (2010)
5. 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)
6. 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)
7. 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)
8. 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)
9. Hauser, J.R.: Handling floating-point exceptions in numeric programs. ACM Trans. Program. Lang. Syst. **18**(2), 139–174 (Mar 1996)
10. IEEE: 754-2008 - IEEE Standard for floating point arithmethic (2008)
11. Magron, V.: Interval Enclosures of Upper Bounds of Roundoff Errors Using Semidefinite Programming. ACM Trans. Math. Softw. **44**(4), 41:1–41:18 (2018)
12. 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)
13. 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)
14. 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)
15. 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)
16. 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)

17. Narkawicz, A., Muñoz, C.: A formally verified generic branching algorithm for global optimization. In: Cohen, E., Rybalchenko, A. (eds.) Verified Software: Theories, Tools, Experiments. pp. 326–343. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
18. 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)
19. 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)
20. Sterbenz, P.H.: Floating Point Computation. Prentice-Hall (1974)
21. The Coq Development Team: The Coq proof assistant reference manual (2020), https://coq.inria.fr, version 8.11.2
22. 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)
23. Zitoun, H.: Search strategies for solving constraint systems over floats for program verification. Theses, Université Côte d'Azur (Oct 2018)
24. 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)