Towards a constraint system for round-off error analysis of floating-point computation

Rémy Garcia (student), Claude Michel (advisor), Marie Pelleau, and Michel Rueher (co-authors)

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

Abstract. In this paper, we introduce a new constraint solver aimed at analyzing the round-off errors that occur in floating-point computations. Such a solver allows reasoning on round-off errors by means of constraints on ranges of error values. This new solver is built by incorporating in a solver for constraints over the floating-point numbers the domain of errors which is dual to the domain of values. Both domains, the domain of values and the domain of errors, are associated with each variable of the problem. Additionally, we introduce projection functions that filter these domains as well as the mechanisms required for the analysis of errors. Preliminary experiments are encouraging.
Numerous works, which are based on an overestimation of actual errors, try to address similar issues. However, they do not provide critical information to reason on those errors, for example, by computing input values that exercise a given error.
To our knowledge, our solver is the first constraint solver with such reasoning capabilities over round-off errors.

Keywords: floating-point numbers · round-off error · constraints over floating-point numbers · domain of errors

1 Introduction

Floating-point computations induce errors due to rounding operations required to close the set of floating-point numbers. These errors are symptomatic of the distance between the computation over the floats and the computation over the reals. Moreover, they are behind many problems, such as the precision or the numerical stability of floating-point computation. Especially when users omit to take into account the nature of floating-point arithmetic and use it directly like real number arithmetic. Identifying, quantifying and localizing those errors are tedious tasks that are difficult to achieve without tools automating it. A well-known example of computations deviation due to errors on floating-point numbers is Rump's polynomial [15]:

$$333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}$$

where $a = 77617$ and $b = 33096$. The exact value of this expression, computed using the GMP library, is $-\frac{54767}{66192} \approx -0.827396056$.

However, when this expression is evaluated on simple floats with a rounding mode set to the nearest even, the computed result is $\approx -6.3382530011411 \times 10^{29}$, which is far apart from the real value. The difference between these two results, about $-6.3382530011411 \times 10^{29}$, emphasizes the need for round-off error analysis tools.

Floating-point computation errors have been the subject of many works based on an overestimation of actual errors. Let us mention the abstract interpreter Fluctuat [6, 5] that combines affine arithmetic and zonotopes to analyze the robustness of programs over floating-point numbers. Another more recent work, PRECiSA [17, 13], relies on static analysis to evaluate round-off errors in a program. Nasrine Damouche [2] and Eva Darulova [3] have developed techniques for the automatic enhancement of numerical code. Their approach is based on an evaluation of round-off errors to estimate the distance between the expression over floats and the expression over reals. These approaches compute an error estimation which can be refined by splitting the search space into subdomains. However, it is not possible to directly reason on those errors, for example, by computing input values that exercise a given error. In order to overcome this lack of reasoning capabilities and to enhance the analysis of errors, we propose to incorporate in a constraint solver over floats [18, 10, 1, 11, 12], the domain of errors which is dual to the domain of values. Both domains are associated with each variable of the problem. Additionally, we introduce projection functions that filter those domains as well as mechanisms required for the analysis of errors. More precisely, we focus on the analysis of deviation between computations over the floats and computations over the reals.

Our approach is based on interval arithmetic for approximating the domains of errors. In addition, a search applied on reduced domains computes input values that satisfy constraints on errors. We deliberately ignore the possibility of an initial observational error on input data even though initial computational errors are handled. Thus input data are assumed with an initial error of zero. For the sake of simplicity, we restrain ourselves to the four classical arithmetic operations. This simplification permits exact computation of values over reals[1] for both values of expressions and computation of errors. Finally, the rounding mode is left to default, i.e. a rounding to the nearest even.

2 Notation and definitions

2.1 Floating-point numbers

The set of floating-point numbers is a finite subset of the rationals that has been introduced to approximate real numbers on a computer. The IEEE standard for

[1] By using a rational arithmetic library for computation on the reals and down to the memory limit.

floating-point arithmetic [9] defines the format of the different types of floating-point numbers as well as the behavior of arithmetic operations on these floating-point numbers. In the sequel, floating-point numbers are restricted to the more common one i.e. simple binary floating-point numbers represented using 32 bits and double binary floating-point numbers represented using 64 bits.

A binary floating-point number v is represented by a triple (s, e, m) where s is the sign of v, e, its exponent and m, its mantissa. When $e > 0$, v is normalized and its value is given by :

$$(-1)^s \times 1.m \times 2^{e-bias}$$

where the *bias* allows us to represent negative values of the exponent. For instance, for 32 bits floating-point numbers, the size of s is 1 bit, the size of e is 8 bits, the size of m is 23 bits and the *bias* is equal to 127.

x^+ denotes the smallest floating-point number strictly larger than x while x^- denotes the largest floating-point number strictly smaller than x. In other words, x^+ is the successor of x while x^- is its predecessor.

An *Ulp*, which stands for *unit in the last place*, is the distance which separate two consecutive floating-point numbers. However, this definition is ambiguous for floats that are a power of 2 like 1.0: in such a case, and if $x > 0$, then $x^+ - x = 2 * (x - x^-)$. To make this point clear, an explicit formulation of this distance is used whenever required.

3 Quantification of computation deviations

Computation on floating-point numbers is different from computation over real numbers due to rounding operations. Since the set of floating-point numbers is a finite subset of the real, in general, the result of an operation on the floats is not a float. In order to close the set of floating-point numbers for those operations, the result should be rounded to the nearest float according to a direction chosen beforehand.

The IEEE 754 norm [9] defines the behavior of floating-point arithmetic. For the four basic operations, it requires correct rounding, i.e. the result of 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 the floats, namely, $\oplus$, $\ominus$, $\otimes$, and $\oslash$, $\cdot$ being the equivalent operation on the reals, and *round* being the rounding function. This property bounds the error introduced by an operation over floats to $\pm\frac{1}{2}ulp(z)$ for correctly rounded operations with a rounding mode set to round to the nearest even float, which is the most frequent rounding mode.

When the result of an operation on the floats is rounded, it is different from the one expected on the reals. Moreover, each operation that belongs to a complex expression is likely to introduce a difference between the expected result on the reals and the one computed on the floats. Whereas for a given operation, the computed float is optimal in terms of rounding, the accumulation of these

approximations can lead to significant deviations, like the one observed with Rump's polynomial.

Compared to its equivalent over the reals, the deviation of a computation over the floats takes root in each elementary operation. Therefore, it is possible to rebuild it from the composition of each elementary operation behavior. Input variables can come with errors attached due to previous computations. For example, for the variable x, 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 on the reals and on the floats respectively. Contrary to an observational error, e_x is signed. This choice is required to capture correctly specific behaviors of floating-point computations, such as error compensations.

As such, computation deviation due to a subtraction can be formulated as follows: for $z = x \ominus y$, the error on z, e_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

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

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

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

In this formula, the last term, $((x_{\mathbb{F}} - y_{\mathbb{F}}) - (x_{\mathbb{F}} \ominus y_{\mathbb{F}}))$, characterizes the error produced by the subtraction operation itself. Let $e_\ominus$ denotes this subtraction operation error. The formula can then be simplified to:

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

The formula comprises two elements: firstly the combination of deviations from input values and secondly, the deviation introduced by the elementary operation.

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

Fig. 1. Computation of deviation for basic operations

Figure 1 formulates computation deviations for all four basic operations. For each of these formulae, the error computation combines deviations from input values and the error introduced by the current operation. Notice that, for multiplication and division, this deviation is proportional to input values.

All these formulae compute the difference between the expected result on the reals and the actual one on the floats for a basic operation. Our constraint solver over the errors on the floats relies on these formulae.

4 Domain of errors

In a classical *CSP*, to each variable x is associated $\mathbf{x}$ its domain of values. It 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:

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

where $\underline{x}_{\mathbb{F}} \in \mathbb{F}$ and $\overline{x}_{\mathbb{F}} \in \mathbb{F}$.

Computation errors form a new dimension to consider. They require a specific domain in view of the distinct nature of elements to represent, but also, due to the possible values of errors which belong to the set of reals. Therefore, we introduce a domain of errors, which is associated with each variable of a problem. Since all arithmetic constraints processed here are reduced to the four basic operations, and since those four operations are applied over floats, i.e. a finite subset of rationals, this domain can be defined as an interval of rationals with bounds in $\mathbb{Q}$:

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

where $\underline{e}_x \in \mathbb{Q}$ and $\overline{e}_x \in \mathbb{Q}$.

Another domain of errors is required for the smooth running of our system: it is the domain of errors on operations, denoted by $e_\odot$, that appears in the computation of the deviations (see Figure 1). Contrary to previous domains, it is not attached to each variable of a problem but to each *instance* of an arithmetic operation of a problem.

Like the domain of errors attached to a variable, it takes values in the set of rationals. Thus, we have:

$$\mathbf{e}_\odot = [\underline{e}_\odot, \overline{e}_\odot] = \{e_\odot \in \mathbb{Q}, \underline{e}_\odot \leq e_\odot \leq \overline{e}_\odot\}$$

where $\underline{e}_\odot \in \mathbb{Q}$ and $\overline{e}_\odot \in \mathbb{Q}$.

This triple, composed of the domain of values, the domain of errors, and the domain of errors on operations, is required to represent the set of phenomena due firstly to possible values on variables and secondly, to different errors that come into play in computation over floats.

5 Projection functions

The filtering process of our solver is based on classical projection functions to reduce the domains of variables. Domains of values can be computed by projection functions defined in [11] and extended in [1] and [10] but new ones are required for the domains of errors.

Those projections on the domains of errors are made through an extension over intervals of formulae from Figure 1. Since these formulae are written over reals, they can naturally be extended to intervals. For example, in the case of the subtraction, we get the four projection functions below:

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

where $\mathbf{e}_x$, $\mathbf{e}_y$, and $\mathbf{e}_z$ are the domains of errors of variables x, y, and z respectively and $\mathbf{e}_\ominus$ is the domain of errors on the subtraction.

Figure 2 gives projection functions for the three other arithmetic operations. Note that the projection on $y_\mathbb{F}$ for the division requires solving a quadratic equation and requires the computation of a square root. Thanks to outward roundings, correct computation of such a square root on rationals is obtained using floating-point square root at the price of an over-approximation. Note also that these projections handle the general case. For the sake of simplicity, special cases like a division by zero are not exposed here.

Addition :

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

Multiplication :

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

Division :

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

with

$$\delta_1 \leftarrow \frac{\mathbf{e}_x - (\mathbf{e}_z - \mathbf{e}_\oslash)\mathbf{e}_y - \sqrt{\mathbf{\Delta}}}{2(\mathbf{e}_z - \mathbf{e}_\oslash)}$$
$$\delta_2 \leftarrow \frac{\mathbf{e}_x - (\mathbf{e}_z - \mathbf{e}_\oslash)\mathbf{e}_y + \sqrt{\mathbf{\Delta}}}{2(\mathbf{e}_z - \mathbf{e}_\oslash)}$$
$$\mathbf{\Delta} \leftarrow [0, +\infty) \cap ((\mathbf{e}_z - \mathbf{e}_\oslash)\mathbf{e}_y - \mathbf{e}_x)^2 + 4(\mathbf{e}_z - \mathbf{e}_\oslash)\mathbf{e}_y\mathbf{x}_\mathbb{F}$$

Fig. 2. Projection functions of arithmetic operation

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. Since the error is not involved in comparison operators, their projection functions only manage domains of values.

The set of those projection functions is used to reduce all variables' domains until a fixed point is reached. For the sake of efficiency, but also to get around potential slow convergence, the fixed point computation is stopped when no domain reduction is greater than 5%.

6 Links between the domain of values and the domain of errors

In order to take advantage of domain reductions of one domain in another domain, clear and strong links must be established between the domain of values and the domain of errors. What naturally occurs for domains of values thanks to constraints on values requires more attention when it comes to the relations between dual domains.

A first relation between the domain of values and the domain of errors on operations is based upon the IEEE 754 norm, which guarantees that basic arithmetic operations are correctly rounded. Since the four basic operations are correctly rounded to the nearest even float, we have

$$(x \odot y) - \frac{(x \odot y) - (x \odot y)^-}{2} \leq (x \cdot y) \leq (x \odot y) + \frac{(x \odot y)^+ - (x \odot y)}{2}$$

where x^- and x^+ denote respectively, the greatest floating-point number strictly smaller than x and the smallest floating-point number strictly larger than x. In other words, the result over floats is, at a half-ulp, the distance between two successive floats from the result over reals. Thus, the error on an operation is contained in this ulp:

$$-\frac{(x \odot y) - (x \odot y)^-}{2} \leq e_\odot \leq +\frac{(x \odot y)^+ - (x \odot y)}{2}$$

This equation sets a relation between the domain of values and the domain of errors on operations: operation errors can never be greater than the greatest half-ulp of the domain of values on the operation result. The projection function for the domain of errors on operations is obtained by extending this formula to intervals:

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

Finally, these links are refined by means of other well-known properties of floating-point arithmetic like the Sterbenz property of the subtraction [16] or the Hauser property on the addition [7]. Both properties give conditions under which these operations produce exact results. As is the well-known property that $2^k * x$ is exactly computed provided that no overflow occurs.

7 Constraints over errors

Usually, constraints available in a solver establish relations between variables of a problem. The duality of domains available in our solver requires introducing a distinction between the domain of values and the domain of errors. In order to preserve the current semantic of expressions, variables keep on representing possible values. A dedicated function, $err(x)$, makes it possible to express constraints over errors. For example, $abs(err(x)) \geq \epsilon$, denote a constraint which demands that the error on variable x be, in absolute value, greater or equal to ϵ. It should be noted that since errors are taking their values in $\mathbb{Q}$, the constraint is over rationals.

When a constraint involves errors and variables, the latter, with domains over floats, are promoted to rationals. Therefore, the constraint is converted to a constraint over rationals.

8 Preliminary experiments

Projection functions and constraints over errors are being evaluated in a prototype based on Objective-CP [8], which already handles constraints over floats thanks to the projection functions of FPCS [12]. All experiments are carried out on a MacBook Pro i7 2,8GHz with 16GB of memory.

8.1 Predator prey

Predator prey [4] has been extracted from the FPBench test suite[2]:

```
double predatorPrey(double x) {
  double r = 4.0;
  double K = 1.11;
  double z = (((r * x) * x) / (1.0 + ((x / K) * (x / K))));
  return z;
}
```

When $x \in [\frac{1}{10}, \frac{3}{10}]$, Objective-CP using a simple and single filtering process reduces the domain of z to $[3.72770587e{-}02, 3.57101682e{-}01]$ and its error domain to $[-1.04160431e{-}16, 1.04160431e{-}16]$. Fluctuat reduces the domain of z to $[3.72770601e{-}02, 3.44238968e{-}01]$ and its error to $[-1.33729201e{-}16, 1.33729201e{-}16]$. Thus, while Fluctuat provides better bounds for the domain of values, Objective-CP provides better bounds for the domain of error. Moreover, our solver can use a search procedure to compute error values that are reachable. For example, we can search for input values such that the error on z will be strictly greater than zero.

With this constraint the solver output $z = 3.35493528698854015512864 6e{-}01$ with an error of $3.09661231429390630181643 2e{-}17$.

[2] See fpbench.org.

Since rational numbers are used for computations of errors it is crucial to take solving time into account. Table 1 shows times in seconds for the generation of round-off error bounds on some benchmarks from FPBench. Solving time for Objective-CP are correct, especially as a search procedure is done in addition of filtering. Those times comfort us in the use of rational numbers for representing errors.

	Gappa	Fluctuat	Real2Float	FPTaylor	PRECiSA	Objective-CP
carbonGas	0.152	**0.025**	0.815	1.209	3.830	*0.060*
verhulst	*0.034*	0.043	0.465	0.812	0.789	**0.032**
predPrey	0.052	**0.031**	0.735	0.916	0.477	*0.050*
turbine1	*0.165*	**0.028**	67.960	2.906	110.272	0.232

Table 1. Times in seconds for the generation of round-off error bounds. For Objective-CP a searh is also used. (bold indicates the best approximation and italic indicates the second best)

9 Conclusion

In this paper, we introduced a constraint solver capable of reasoning over computation errors on floating-point numbers. It is built over a system of dual domains, the first one characterizing possible values that a variable of the problem can take and the second one defining errors committed during computations. Moreover, there are particular domains, bind to instances of arithmetic operations in numerical expressions of the constraints, which represent errors in those operations. Our solver, enhanced with projection functions and constraints over errors, offers unique possibilities to reason on computation errors. Preliminary experiments are promising and will naturally be reinforced with more benchmarks.

Such a solver might appear limited by the use of rational numbers and multiprecision integers. However, a thorough examination of the solver behavior has shown that its main limit lies in its approximation of round-off errors. Firstly, round-off errors are not uniformly distributed across input values. As a result, finding input values that satisfy some error constraints has often to resort to an enumeration of possible values. Secondly, round-off error of operations are overestimated. Such an overestimation does not perform the fine domain reductions that would allow an efficient search. Therefore, a deeper understanding and a tighter representation of the round-off error on an operation basis is a must to actually improve the behavior of our solver.

Further work include extending support for a wider set of arithmetic functions, improving the search to quickly find solutions in the presence of constraints over errors and to add global optimization capabilities, for example, by using a branch-and-bound method. Then, by formulating a problem as an optimization problem, we should be in a position to determine for which input values the error is maximal.

Another direction of improvement is the combination of CSP with other tools dedicated to round-off error like abstract interpreter in an approach similar to what has already been done for domains of values [14].

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. Damouche, N., Martel, M., Chapoutot, A.: Improving the numerical accuracy of programs by automatic transformation. STTT **19**(4), 427–448 (2017)
3. 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)
4. Darulova, E., Kuncak, V.: Sound compilation of reals. pp. 235–248. POPL '14 (2014)
5. Ghorbal, K., Goubault, E., Putot, S.: A logical product approach to zonotope intersection. In: Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19,. LNCS, vol. 6174, pp. 212–226 (2010)
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. Hauser, J.R.: Handling floating-point exceptions in numeric programs. ACM Trans. Program. Lang. Syst. **18**(2), 139–174 (Mar 1996)
8. Hentenryck, P.V., Michel, L.: The objective-cp optimization system. In: 19th International Conference on Principles and Practice of Constraint Programming (CP 2013). pp. 8–29 (2013)
9. IEEE: 754-2008 - IEEE Standard for floating point arithmethic (2008)
10. 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)
11. 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)
12. 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)
13. Moscato, M.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 - 36th International Conference, SAFECOMP 2017, Trento, Italy, September 13-15. pp. 213–229 (2017)
14. Ponsini, O., Michel, C., Rueher, M.: Verifying floating-point programs with constraint programming and abstract interpretation techniques. Automated Software Engineering **23**(2), 191–217 (Jun 2016)
15. Rump, S.: Algorithms for verified inclusions: Theory and practice. In: Moore, R.E. (ed.) Reliability in Computing: The Role of Interval Methods in Scientific Computing, pp. 109–126. Academic Press Professional, Inc., San Diego, CA, USA (1988)
16. Sterbenz, P.H.: Floating Point Computation. Prentice-Hall (1974)
17. 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)
18. 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)