Rigorous Floating-Point Round-Off Error Analysis in PRECiSA 4.0

Laura Titolo[1][0000-0001-7820-7640]✉, Mariano Moscato[1][0000-0002-6468-9498], Marco A. Feliu[1][0009-0002-6943-9479], Paolo Masci[1][0000-0002-0667-7763], and César A. Muñoz[2][0000-0002-8503-5514]

[1] Analytical Mechanics Associates Inc.
{`laura.titolo, mariano.m.moscato, marco.feliu, paolo.m.masci`}`@nasa.gov`
[2] NASA Langley Research Center,
`cesar.a.munoz@nasa.gov`

Abstract. Small round-off errors in safety-critical systems can lead to catastrophic consequences. In this context, determining if the result computed by a floating-point program is accurate enough with respect to its ideal real-number counterpart is essential. This paper presents PRECiSA 4.0, a tool that rigorously estimates the accumulated round-off error of a floating-point program. PRECiSA 4.0 combines static analysis, optimization techniques, and theorem proving to provide a modular approach for computing a provably correct round-off error estimation. PRECiSA 4.0 adds several features to previous versions of the tool that enhance its applicability and performance. These features include support for data collections such as lists, records, and tuples; support for recursion schemas; an updated floating-point formalization that closely characterizes the IEEE-754 standard; an efficient and modular analysis of function calls that improves the performances for large programs; and a new user interface integrated into Visual Studio Code.

1 Introduction

Round-off errors arise from the difference between real numbers and their finite precision representations. In a floating-point program, round-off errors accumulate throughout the computation. This may lead to a large divergence between the result computed using floating-point arithmetic and the one ideally obtained using real-number arithmetic. In application domains such as avionics, even small rounding errors may have catastrophic consequences if they are not carefully accounted for. Examples of these errors have been found, for instance, in geofencing applications [29] and position encoding algorithms [41]. Several tools have been proposed over the years to reason about floating-point errors (see [8] for an overview). However, most of the proposed tools either target straight-line code and scalar values only, or do not provide sufficient formal guarantees. This limits their applicability to safety-critical real-world applications.

This paper presents PRECiSA 4.0, an open-source[3] tool for automatic floating-point round-off error analysis. PRECiSA computes a sound and accurate estima-

[3] `https://github.com/nasa/PRECiSA`.

tion of the round-off error that may occur in a floating-point program. It supports a large variety of mathematical operators and programming language constructs, including conditionals, let-in expressions, and bounded loops. In addition, PRECiSA automatically generates formal certificates that can be externally checked in the Prototype Verification System (PVS) [33]. These certificates provide formal guarantees on the soundness of the computed round-off error bounds.

An overview of PRECiSA is presented in Section 2. PRECiSA 4.0 adds the following features with respect to previous versions of the tool [28,39]:

- A novel *modular analysis* for *function calls* has been implemented (Section 3). The user can choose to apply an abstraction on the computed round-off error expressions for function calls to speed up the analysis execution time. This abstraction has been shown to be effective in the analysis of large programs with multiple function calls.
- Support for *data collections* such as lists, records, and tuples has been implemented. In addition, to operate on these collections, native support for the *map* and *fold* recursion schemas has been added (Section 4). This new feature avoids the task of manually unfolding the program, resulting in a less error-prone and more efficient analysis of data collections.
- Support for a new *floating-point formalization* has been added. This formalization faithfully characterizes the IEEE-754 standard [24], including special values such as NaN, signed zeros, and infinities (Section 5).
- VSCode-PRECiSA, a new user interface integrated into Visual Studio Code, has been added to the PRECiSA distribution (Section 6). Besides providing an intuitive way of presenting the analysis results, VSCode-PRECiSA automatizes and simplifies different kinds of tasks such as comparative, sensitivity, and interval analysis. In addition, VSCode-PRECiSA features a new graphical visualization of the values that may cause *conditional instability.* This phenomenon occurs when floating-point round-off errors impact the evaluation of Boolean expressions in conditional guards, thus affecting the control-flow of a program.

In addition to presenting these new features, an experimental evaluation comparing PRECiSA with other state-of-the-art tools is presented in Section 7 together with a discussion on related work.

2 PRECiSA

The round-off error of the floating-point expression $\widetilde{op}(\tilde{v}_i)_{i=1}^n$ with respect to the real-valued expression $op(r_i)_{i=1}^n$, where $\widetilde{op}$ is a floating-point operator representing a real-valued operator op and $\tilde{v}_i$ is a floating-point value representing a real value r_i, for $1 \leq i \leq n$, is given by a combination of (i) the propagation of the errors carried out by the arguments $\tilde{v}_i$, and (ii) the error introduced by the application of $\widetilde{op}$ versus op. Throughout this paper, floating-point variables, operands, and expressions are denoted with a tilde on top. The IEEE-754 standard [24] states that every basic operation should be performed as if it were

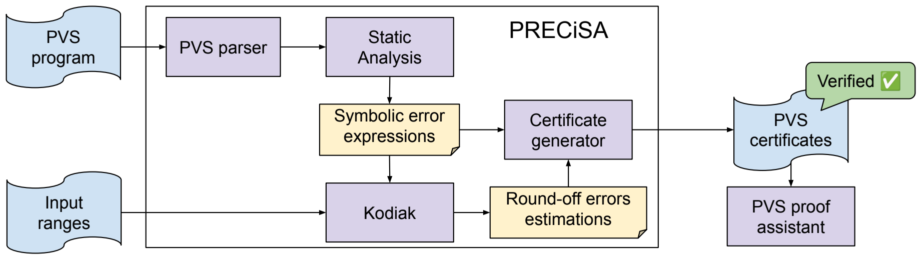


Fig. 1: PRECiSA workflow.

calculated with infinite precision and then rounded to the nearest floating-point value. Thus, the following inequality is assumed to hold for an n-ary floating-point operator $\widetilde{op}$.

$$|R(\widetilde{op}(\tilde{v}_i)_{i=1}^n) - op(\mathrm{R}(\tilde{v}_i))_{i=1}^n| \leq \tfrac{1}{2} ulp\,(op(\mathrm{R}(\tilde{v}_i))_{i=1}^n)\,, \tag{2.1}$$

where R is the projection from floats to reals, and the function $ulp\,(r)$ (unit in the last place), for a given real number r, measures the distance between the two consecutive floating-point numbers f_1 and f_2 such that $f_1 < r \leq f_2$. The round-off error of a real-valued expression can be bounded as

$$|R(\widetilde{op}(\tilde{v}_i)_{i=1}^n) - op(r_i)_{i=1}^n| \leq \varepsilon_{op}(r_i, e_i)_{i=1}^n + \tfrac{1}{2} ulp\,(op(\mathrm{R}(\tilde{v}_i))_{i=1}^n)\,, \tag{2.2}$$

where $\varepsilon_{op}(r_i, e_i)_{i=1}^n$ represents an overestimation of the difference between the application of the real operator on real values and the application of the same operator on the floating-point arguments, and each e_i is a positive real-valued expression modeling an upper bound of the round-off error carried by the floating-point arguments $\tilde{v}_i$ representing the real-valued expression r_i, i.e., $|\mathrm{R}(\tilde{v}_i) - r_i| \leq e_i$.

PRECiSA assumes compliance with the IEEE-754 standard and uses the round-off error model of Formula (2.2) for correctly rounded operators. Dedicated error approximations are defined for a wide variety of mathematical operators, including arithmetic operators, square root, trigonometric functions, exponential and logarithmic functions, floor, and ceiling. For each of these operators, an error expression $\epsilon_{op}(r_i, e_i)_{i=1}^n$ is defined as a function of the real-valued operands and corresponding errors such that

$$\epsilon_{op}(r_i, e_i)_{i=1}^n \geq \varepsilon_{op}(r_i, e_i)_{i=1}^n + \tfrac{1}{2} ulp\,(op(\mathrm{R}(\tilde{v}_i))_{i=1}^n)\,. \tag{2.3}$$

PRECiSA accepts as input a floating-point program P, which consists of a set of function declarations in the language of PVS, and initial ranges for the input variables, and it computes a correct overestimation of the round-off error that may occur for each function in P. An overview of the PRECiSA workflow is depicted in Fig. 1. PRECiSA first performs a static analysis on the input program by computing the abstract semantics defined in [39]. For every function declaration $\tilde{f}(\tilde{x}_i)_{i=1}^n$ in the input program, PRECiSA computes a set of *conditional error* *bounds* of the form $\langle \eta, \widetilde{\eta} \rangle \twoheadrightarrow (r, e)$, where η is a Boolean expression on reals, $\widetilde{\eta}$ is a Boolean expression on floats, and r, e are real-valued symbolic arithmetic expressions. Intuitively, $\langle \eta, \widetilde{\eta} \rangle \twoheadrightarrow (r, e)$ indicates that if the conditions η and $\widetilde{\eta}$ are satisfied, the result of evaluating $\widetilde{f}(\tilde{x}_i)_{i=1}^n$ using exact real-number arithmetic is r and the round-off error of the floating-point implementation is bounded by e. The error expression e is built compositionally following the structure of the function body. The Boolean expressions η and $\tilde{\eta}$ model the information on the control flow of the program (i.e., the path conditions from the if-then-else constructs) and the additional restrictions needed when the operators are not total. For example, when dealing with the division operation, it is necessary to guarantee that the divisor is not zero.

The static analysis collects information about real and floating-point execution paths separately. Thus, it is possible to quantify the error due to the so-called *unstable conditions*. This phenomenon occurs when the Boolean guard of a conditional statement is affected by round-off errors. In this case, the real and floating-point Boolean evaluation may be different, causing the control-flow of the floating-point implementation to diverge with respect to its ideal real-number counterpart. An abstraction technique has been introduced in [39] to mitigate the state explosion resulting from the sound treatment of conditional statements. This abstraction collapses the information on the conditional error bounds by keeping separated stable and unstable cases. This way, the accuracy of the error analysis is preserved while the size of the state space is reduced.

Example 1. Consider the function $\widetilde{tcoa}$ that computes the time to co-altitude of two aircraft whose relative altitude is $\tilde{s}$ and relative vertical speed is $\tilde{v}$.

$$\widetilde{tcoa}(\tilde{s}, \tilde{v}) = \texttt{if } \tilde{s} \mathbin{\tilde{*}} \tilde{v} < 0 \texttt{ then } -(\widetilde{\tilde{s}/\tilde{v}}) \texttt{ else } -1, \tag{2.4}$$

PRECiSA computes a set of four different conditional error bounds:

$$\begin{aligned}
&\{\langle s * v < 0 \land v \neq 0, \tilde{s} \mathbin{\tilde{*}} \tilde{v} < 0 \land \tilde{v} \neq 0 \rangle \twoheadrightarrow (-s/v, \epsilon_{/}(s, v, e_s, e_v)), && (2.5)\\
&\ \langle s * v \geq 0, \tilde{s} \mathbin{\tilde{*}} \tilde{v} \geq 0 \rangle \twoheadrightarrow (-1, 0), && (2.6)\\
&\ \langle s * v < 0 \land v \neq 0, \tilde{s} \mathbin{\tilde{*}} \tilde{v} \geq 0 \rangle \twoheadrightarrow (-s/v, |-s/v - 1|), && (2.7)\\
&\ \langle s * v \geq 0 \land v \neq 0, \tilde{s} \mathbin{\tilde{*}} \tilde{v} < 0 \land \tilde{v} \neq 0 \rangle \twoheadrightarrow (-1, |s/v - 1 + \epsilon_{/}(s, v, e_s, e_v)|)\}. && (2.8)
\end{aligned}$$

The real-valued variables s and v represent the real values of $\tilde{s}$ and $\tilde{v}$, respectively, while e_s and e_v are two positive real variables representing the round-off error of $\tilde{s}$ and $\tilde{v}$, respectively. Formula (2.5) and Formula (2.6) correspond to the cases where real and floating-point computational flows coincide. In Formula (2.5), the negation operator does not contribute to the rounding error, and the following symbolic round-off error expression is computed for the division:

$$\epsilon_{/}(s, v, e_s, e_v) = \frac{|v| e_s + |s| e_v}{v^2 - |v| e_v} + \tfrac{1}{2} ulp\left(\frac{|s| + e_s}{|v| - e_v}\right). \tag{2.9}$$

In Formula (2.6), the error is 0 since the output is an integer constant. Formula (2.7) and Formula (2.8) model the unstable paths. In these cases, the error is the difference between the output of the two branches taking into account the round-off error of the floating-point result.

The described static analysis is purely compositional, i.e., no assumption is made on the values of the input variables, and the error expressions are composed in a modular fashion. Given initial ranges for the input variables, PRECiSA uses Kodiak[4], a rigorous global optimizer, to compute a sound enclosure of the maximum of the symbolic error expression e. Kodiak implements a formally verified branch-and-bound algorithm presented in [32]. This branch-and-bound algorithm relies on enclosure functions for mathematical operators. These enclosure functions compute provably correct over- and under- approximations of the symbolic error expressions using either interval arithmetic or Bernstein polynomial basis. The algorithm recursively splits the domain of the function into smaller subdomains and computes an enclosure of the original expression in these subdomains. The recursion stops when a precise enclosure is found, based on a given precision, or when a given maximum recursion depth is reached. Both precision and maximum recursion depth can be specified as parameters in PRECiSA. Increasing the value of these parameters will likely improve the accuracy of the analysis but may also increase the execution time. The output of Kodiak is a numerical enclosure for each symbolic error expression. When a function $\tilde{f}$ is associated with more than one conditional error bound, e.g., in the case of conditionals, the overall round-off error of $\tilde{f}$ is defined as the maximum of all the error expressions.

To provide formal guarantees on the analysis results, PRECiSA generates proof certificates ensuring that the round-off error estimations are correct. PRECiSA relies on the higher-order logic interactive theorem prover PVS and a floating-point round-off error formalization included in the NASA PVS Library.[5] More details on this formalization will be given in Section 5. For each function, the information in the conditional error bounds is encoded as a PVS lemma stating that, provided the conditions are satisfied and the input variables are in the given numerical ranges, the difference between the floating-point implementation and the real-number specification is at most the computed error estimation. Automatic strategies have been implemented to check the symbolic error expression correctness and the enclosure computed by Kodiak by executing the formally verified PVS implementation of the branch-and-bound algorithm of [32].

3 Optimized Modular Function Call Analysis

PRECiSA supports the compositional analysis of non-recursive function calls. The analysis works by computing a set of conditional error bounds for each function declaration and building an *interpretation* I mapping each function to its semantics. When the analysis encounters a function call $\tilde{f}(\tilde{x}_i)_{i=1}^{n}$, it performs a look-up in the interpretation I. For each conditional error bound associated

[4] `https://github.com/nasa/Kodiak`.
[5] `https://github.com/nasa/pvslib`.

to the function $\langle \phi, \tilde{\phi} \rangle_t \twoheadrightarrow (r, e) \in I(\tilde{f}(\tilde{x}_i)_{i=1}^n)$, PRECiSA performs a substitution of the formal parameters with the actual ones, computing all the possible combinations. For each actual parameter and each conditional error bound in its semantics, $\langle \phi_i, \tilde{\phi}_i \rangle_{t_i} \twoheadrightarrow (r_i, e_i)$, the following conditional error bound is computed for the function call

$$\langle \phi' \wedge \bigwedge_{i=1}^{n} \phi_i, \tilde{\phi}' \wedge \bigwedge_{i=1}^{n} \tilde{\phi}_i \rangle \twoheadrightarrow (r', e'),$$

where $r' = r[\tilde{x}_i/r_i]_{i=1}^n$, $e' = e[\epsilon_{\tilde{x}_i}/e_i]_{i=1}^n$,$\phi' = \phi[\tilde{x}_i/r_i, \epsilon_{\tilde{x}_i}/e_i]_{i=1}^n$, and $\tilde{\phi}' = \tilde{\phi}[\tilde{x}_i/r_i, \epsilon_{\tilde{x}_i}/e_i]_{i=1}^n$. More details on the semantics can be found in [39].

This approach guarantees correctness and accuracy for the optimization process since the error expressions of each function call and of its arguments are unfolded in the global error expression. However, such an error expression may become extremely large for programs with multiple and nested function calls.

To overcome this problem, PRECiSA 4.0 implements an alternative abstract semantics for function calls. In this approach, during the symbolic analysis process, when a function call $\tilde{f}(\tilde{x}_i)_{i=1}^n$ is encountered, instead of computing all the combinations and unfolding the semantics of the function, a placeholder is placed in the call site. Then, during the optimization phase of the analysis, this placeholder is replaced with the worst-case round-off error for the function $\tilde{f}$, computed by Kodiak by optimizing the error expression associated to $\tilde{f}$ and obtained from the interpretation I. It is crucial that the global optimization is executed at each calling site with the correct input ranges for the function arguments with respect to the initial range inputs, provided by the user, and the accumulated round-off error of the arguments. To enable this, PRECiSA relies again on the global optimizer Kodiak. For each argument $\widetilde{arg}$, Kodiak computes its range $[l, u]$ by optimizing the real-valued counterpart of the argument expression. To improve efficiency without compromising too much precision, plain interval arithmetic with no branching—setting the maximum depth parameter to 1 in Kodiak—is used in this phase. The symbolic round-off error expression associated to $\widetilde{arg}$ is computed by PRECiSA. The error due to unstable branches is also taken into account in this phase. This error expression is maximized by Kodiak, and the result err is used to enlarge the argument ranges, obtaining $[l - err, u + err]$. This range is the one used to maximize the function's error expression for that specific call site. The symbolic error expression for each function is computed just once when the interpretation I is built and then its numerical value is computed by maximizing it with different input ranges. The proposed abstract semantics may lead to a loss of correlation between the variables and, potentially, to less accurate estimations. Depending on the desired accuracy/efficiency threshold, the user can choose to perform this abstract function call analysis (the default behavior in PRECiSA 4.0) or to use the option that unfolds the semantics of the function calls and arguments.

The optimization of function calls was key for performing a formal analysis of the NASA DAIDALUS library [30]. This library provides a reference implementation of detect-and-avoid capabilities for unmanned aircraft systems intended to keep aircraft safely separated. In [5], the application of a toolchain to extract a formally verified floating-point C implementation of a DAIDALUS module is presented. The extracted code is annotated with program contracts modeling how the round-off error accumulates through the computation and is instrumented to detect conditional instability. PRECiSA is used in this toolchain as a library to compute round-off errors following the approach presented in [42]. To successfully apply the previous version of PRECiSA to the DAIDALUS module, a pre-processing of the input specification was needed. Without this pre-processing, which included a program slicing and several semantics-preserving simplifications, PRECiSA did not terminate after several minutes. This was due to the complexity of the module which features several conditionals, predicates, and function calls. Using the analysis optimization described in this section, the new version of PRECiSA is able to analyze the original DAIDALUS module without the slicing and simplification used in [5]. Fig. 2 and Table 1 show the comparison between the original and the abstract analysis for the numerical functions in the DAIDALUS module. In this case study, the function abstraction improves the performance of the analysis without sacrificing the accuracy. In some cases, slightly more accurate estimations are obtained. This may be due to the large size of the unfolded error expressions, for which the branch-and-bound may not be able to reach enough accuracy within the specified maximum depth. For instance, for `vertical_WCV` (see [30]), the unfolding process times out after 5 minutes.

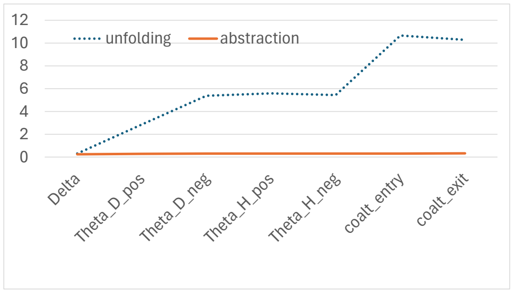


Fig. 2: Times in seconds for the analysis of the DAIDALUS module.

	unfolding	abstraction
Delta	4.69-14	4.68E-14
Theta_D_pos	1.29E-07	1.07E-07
Theta_D_neg	1.29E-07	1.07E-07
Theta_H_pos	3.55E-15	3.55E-15
Theta_H_neg	3.55E-15	3.55E-15
coalt_entry	8.88E-15	6.66E-15
coalt_exit	3.77E-15	3.77E-15
vertical_WCV	time-out	1.77E-15

Table 1: Experimental results on the round-off error of the DAIDALUS module.

4 Data Collections and Bounded Recursion Support

Previous versions of PRECiSA, as well as the majority of floating-point error analysis tools, focus on scalar values. However, it is often the case that safety-critical numerical code makes use of data structures such as lists, tuples, and records. For instance, in the NASA-developed libraries DAIDALUS [30] (aircraft detect-and-avoid) and PolyCARP [31] (polygon computations), a point in space is represented as a tuple (x, y); polygons, used to represent keep-in and keep-out areas such as geofences and weather cells, are represented as lists of points; and aircraft position and velocity vectors are represented as records. These libraries also use bounded loops and typical functional language recursion structures such as map and fold. To enhance the applicability of PRECiSA to these libraries of interest to NASA, support for data collections and the bounded recursion schemas *map* and *fold* have been added to PRECiSA.

Data collections are admitted both as arguments and as return types of functions. Records and tuples are treated in a similar way in PRECiSA. The variable environment used by PRECiSA to store the semantics of local and input variables has been enhanced to accommodate record fields and tuple indices. When a function returns a record or a tuple, PRECiSA performs the static analysis for each element, thus the result is a record or tuple of round-off errors. Furthermore, the structure of the function interpretation I has been modified to support fields and indices. When a function of type record or tuple is called by another function and a field or index is accessed, a lookup in the interpretation is performed as expected.

In contrast to records and tuples, the round-off error of a list is defined as the maximum of the errors of its elements assuming that they are in the same given input range. PRECiSA 4.0 adds support for the following recursion schemas that operate on lists.

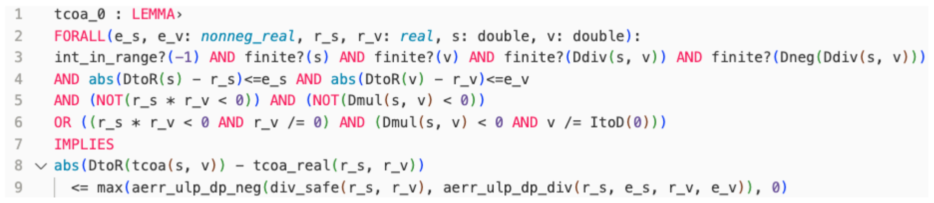
$$\texttt{map}\ \tilde{f}\ [l_1, \ldots, l_n] = [\tilde{f}(l_1), \ldots, \tilde{f}(l_n)], \tag{4.1}$$

$$\texttt{fold}\ \tilde{f}\ a\ [l_1, \ldots, l_n] = \tilde{f}(l_1, \ldots (\tilde{f}(l_{n-1}, \tilde{f}(l_n, a))) \ldots). \tag{4.2}$$

Instead of unrolling the definitions and computing a large error expression, it is sufficient to retrieve the error expression associated to function $\tilde{f}$ in the interpretation, and apply the global optimization process with the correct input variable ranges. For the *map* schema this process is straightforward since all elements in a list are assumed to be in the same input range provided by the user. For the *fold* schema, similar to the function call analysis presented in Section 3, it is possible to compute an overestimation of the input ranges in Kodiak. In this phase, n branch-and-bound evaluations of $\tilde{f}$ are performed, where n is the length of the list. The symbolic error expression for $\tilde{f}$ is computed once and then maximized for different values.

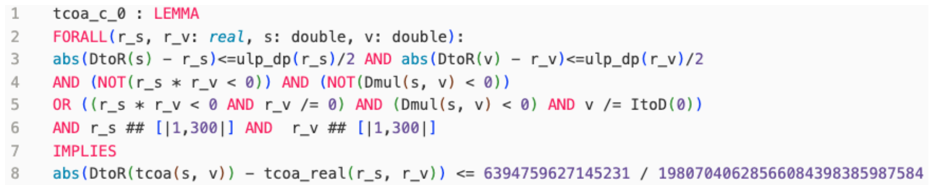
5 Floating-point Formalization

In addition to computing error bounds, an important feature of PRECiSA is the generation of PVS proof certificates that formally ensure that these bounds are correct. PRECiSA relies on the higher-order logic interactive theorem prover PVS [33] and a floating-point formalization originally presented in [6] and extended in [28]. This formalization includes basic definitions related to floating-point numbers, such as their representation, the notion of *ulp*, the notion of subnormal float, and the definition of correctly rounded operators. In addition,



```
1    tcoa_0 : LEMMA›
2    FORALL(e_s, e_v: nonneg_real, r_s, r_v: real, s: double, v: double):
3    int_in_range?(-1) AND finite?(s) AND finite?(v) AND finite?(Ddiv(s, v)) AND finite?(Dneg(Ddiv(s, v)))
4    AND abs(DtoR(s) - r_s)<=e_s AND abs(DtoR(v) - r_v)<=e_v
5    AND (NOT(r_s * r_v < 0)) AND (NOT(Dmul(s, v) < 0))
6    OR ((r_s * r_v < 0 AND r_v /= 0) AND (Dmul(s, v) < 0 AND v /= ItoD(0)))
7    IMPLIES
8  ⌄ abs(DtoR(tcoa(s, v)) - tcoa_real(r_s, r_v))
9    |  <= max(aerr_ulp_dp_neg(div_safe(r_s, r_v), aerr_ulp_dp_div(r_s, e_s, r_v, e_v)), 0)
```

Fig. 3: Symbolic error lemma in PVS for the $\widetilde{tcoa}$ function.



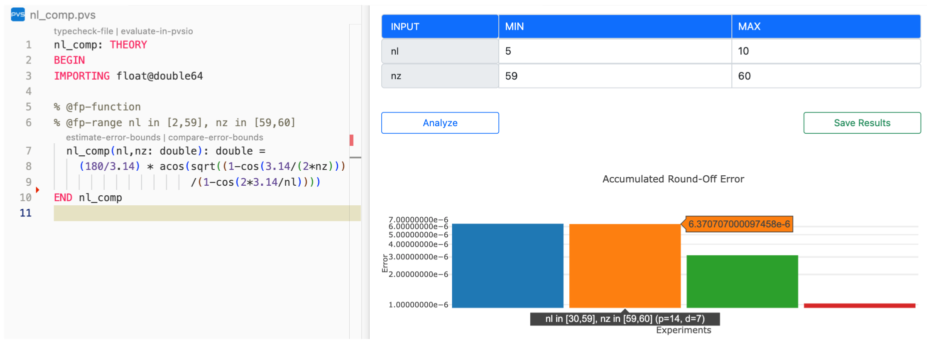
```
1    tcoa_c_0 : LEMMA
2    FORALL(r_s, r_v: real, s: double, v: double):
3    abs(DtoR(s) - r_s)<=ulp_dp(r_s)/2 AND abs(DtoR(v) - r_v)<=ulp_dp(r_v)/2
4    AND (NOT(r_s * r_v < 0)) AND (NOT(Dmul(s, v) < 0))
5    OR ((r_s * r_v < 0 AND r_v /= 0) AND (Dmul(s, v) < 0) AND v /= ItoD(0))
6    AND r_s ## [|1,300|] AND  r_v ## [|1,300|]
7    IMPLIES
8    abs(DtoR(tcoa(s, v)) - tcoa_real(r_s, r_v)) <= 6394759627145231 / 19807040628566084398385987584
```

Fig. 4: Numeric error lemma in PVS for the $\widetilde{tcoa}$ function.

it includes a collection of formally verified round-off error estimations for a wide range of mathematical operators. Since PRECiSA's previous release, the PVS floating-point formalization has been restructured and updated to model closely the IEEE-754 standard. To accommodate this change, the certificate generation and the automated proof strategies have been updated in PRECiSA 4.0.

The previous version of PRECiSA assumed that floating-point values were unbounded, meaning that they could be outside the ranges defined by the IEEE-754 standard. Furthermore, special values such as signed zeros, infinities, and NaN were not represented. The new version explicitly introduces bounds for different architectures and special values as defined in the standard. Thus, all floating-point values are required to be either special values or within a valid range.

As an example, Fig. 3 depicts one of the lemmas generated for the function $\widetilde{tcoa}$ from Example 1. Line 2 quantifies over the floating-point variables, `s` and `v`, their real number counterparts, `r_s` and `r_v`, and the non-negative error variables `e_s` and `e_v`. Line 3 states that all the expressions are finitely representable, thus no overflow or NaN can occur. Line 4 states that `e_s` (resp., `e_v`) over-approximates the difference between `r_s` and `s` (resp., `r_v` and `v`). Lines 5-6 specify the Boolean conditions that model the stable conditional error bounds in Formula (2.5) and Formula (2.6). The consequent of the lemma states that the round-off error of $\widetilde{tcoa}$ is at most the maximum between the error of the division $-(\tilde{s}\tilde{/}\tilde{v})$ and 0, which is the representation error of the value -1. Fig. 4 shows a concrete numerical instantiation of the lemma in Fig. 3, which is also automatically generated by PRECiSA. The input ranges are declared in Line 6. The error computed by Kodiak is shown in Line 8 and roughly corresponds to $3.23E-13$. The generated valid range conditions can be used as implicit overflow detectors. In fact, if the value of an expression cannot be proven to be in the



nl_comp.pvs

```
typecheck-file | evaluate-in-pvsio
nl_comp: THEORY
BEGIN
IMPORTING float@double64

% @fp-function
% @fp-range nl in [2,59], nz in [59,60]
  estimate-error-bounds | compare-error-bounds
  nl_comp(nl,nz: double): double =
    (180/3.14) * acos(sqrt((1-cos(3.14/(2*nz)))
                      /(1-cos(2*3.14/nl))))
END nl_comp
```

INPUT	MIN	MAX
nl	5	10
nz	59	60

Fig. 5: Round-off error analysis in VSCode-PRECiSA.

range, the lemma cannot be proven. This indicates that an overflow may have occurred.

6 VSCode-PRECiSA User Interface

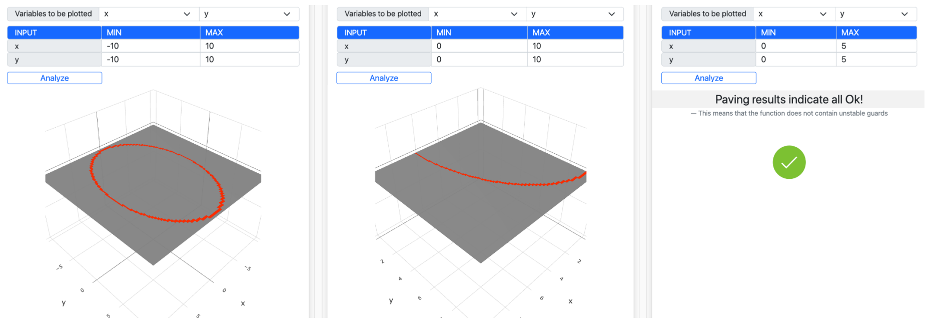
VSCode-PRECiSA[6] implements a graphical user interface that integrates PRECiSA into Visual Studio Code, a widely used software development environment developed by Microsoft. Analysis results from PRECiSA are presented using both a bar chart plot diagram (see Fig. 5) and a numerical table. The table presents the numerical results of the analysis along with information on the instability error measuring the divergence of the conditional branches, if applicable, and specific details about the parameters used for the analysis. A series of analysis experiments can be performed for different ranges of input values and combinations of analysis parameters. VSCode-PRECiSA also provides specialized views that facilitate and automate different tasks typically performed with PRECiSA: interval analysis, sensitivity analysis, comparative analysis, and conditional instability analysis.

The *interval analysis* view divides a range of input values into n equally-sized sub-ranges, where n is a positive natural number provided by the user. Floating-point round-off error estimations are computed for each sub-range. The results obtained in this view can be used to gain insights on how to reimplement functions to minimize their round-off errors.

The *sensitivity analysis* view evaluates the floating-point round-off error of a function when the range of input values is affected by a given uncertainty coefficient provided by the user. This view automates the task of checking the robustness of a program to round-off errors, i.e., whether small variations of a program's input values lead to unexpectedly large variations in the output.

The *comparative analysis* view shows the floating-point round-off error of two functions evaluated on the same input variables. This view facilitates the assessment of the round-off error in two alternative implementations of an algorithm.

[6] `https://github.com/nasa/PRECiSA/tree/master/vscode-precisa`.



Variables to be plotted x y

INPUT	MIN	MAX
x	-10	10
y	-10	10

Analyze

Variables to be plotted x y

INPUT	MIN	MAX
x	0	10
y	0	10

Analyze

Variables to be plotted x y

INPUT	MIN	MAX
x	0	5
y	0	5

Analyze

Paving results indicate all Ok!

— This means that the function does not contain unstable guards

Fig. 6: Conditional instability analysis in VSCode-PRECiSA.

The toolkit automatically feeds the two functions with the same input ranges and the analysis results are displayed side-by-side in a bar chart.

As mentioned in Section 2, PRECiSA estimates the error associated with unstable conditionals and computes the conditions under which the ideal real number path diverges from the floating-point one. These conditions, called *instability conditions*, are represented by sets of Boolean expressions over both real and floating-point numbers. The *conditional instability analysis* in VSCode-PRECiSA presents visual information on these instability conditions, highlighting which combinations of input variables may alter the control flow of a floating-point program with respect to its ideal real number counterpart. A *2D-mesh* plot is created for a selected pair of variables where the red areas correspond to the regions of possible instability. These regions of instability are computed by the branch-and-bound paving functionality of Kodiak. The paver partitions the input space into regions (called boxes) and uses interval arithmetic to compute the value of the instability conditions over each input region. Due to the over-approximation introduced by interval arithmetic, Kodiak classifies every box as "certainly satisfy," "possibly satisfy," and "certainly do not satisfy." The "possibly satisfy" boxes are progressively refined until a maximum refinement depth or a minimum precision (box size) is reached. The set of boxes that "certainly" and "possibly" satisfy the instability conditions form a sound over-approximation of the inputs that may cause unstable behaviors and, as a consequence, may lead to large computation errors. To the best of the authors' knowledge, PRECiSA is the only tool that supports this kind of analysis. Fig. 6 shows the results of the instability analysis for the following function that checks if a point is inside an ellipse-shaped area.

$$\widetilde{pointInEllipse}(\tilde{x},\tilde{y}) = \texttt{if } \tilde{x} \mathbin{\tilde{*}} \tilde{x} \mathbin{\tilde{/}} 4 \mathbin{\tilde{+}} \tilde{y} \mathbin{\tilde{*}} \tilde{y} \mathbin{\tilde{/}} 9 \leq 10 \texttt{ then } 1 \texttt{ else } -1. \tag{6.1}$$

The figure illustrates that unstable tests may occur for values close to the border of the ellipse, though regions of instability are not always as obvious (see [29] for an example).

7 Related Work

Diverse analysis techniques and tools that estimate the round-off error of floating-point computations have been proposed in the literature.

Gappa [16] computes enclosures for floating-point expressions via interval arithmetic that can be checked in the Coq proof assistant. This method enables a quick computation, but may result in pessimistic error estimations. In Gappa, the bound computation, the certification construction, and their verification may require hints from the user. Thus, some level of expertise is required, unlike PRECiSA, which is fully automatic.

Fluctuat [19] is a commercial analyzer that accepts as input a C program with annotations about input bounds and uncertainties, and it produces bounds for the round-off error of the program expressions. Fluctuat uses a zonotopic abstract domain [21] that extends affine arithmetic [17]. It can soundly identify whether unstable conditional may occur [22] and it provides support for iterative programs by using the widening operators introduced in [18,20]. Unlike PRECiSA, Fluctuat does not produce formal certificates. PRECiSA also implements a widening operator [39], which takes advantage of the information contained in the path conditions of the conditional error bounds to determine when the round-off error of a program may converge. This widening has been applied to simple programs where the error is known to stabilize in a few iterations. More work is needed in this direction to define an effective widening operator for estimating round-off errors for recursive programs.

FPTaylor [37] uses symbolic Taylor expansions to approximate floating-point straight-line expressions and, similar to PRECiSA, applies a global optimization technique to obtain numerical enclosures for round-off errors. It provides support for different rounding modalities such as to-the-nearest, toward infinity, and toward zero. Previous versions of FPTaylor emitted certificates for HOL Light [23], however this functionality appears as deprecated in the last release.

Satire [15] is a tool for estimating round-off errors for straight-line floating-point code with a focus on efficiency. It combines a variant of the technique presented in [37] with an abstraction heuristic that replaces parts of the symbolic error expression with pre-computed constants. Similar to the abstraction presented in Section 3, this approach can lead to a loss of correlation between variables and possibly less accurate results, however, it improves the performance of the tool, and it provides a good compromise to scale up to expressions with thousands of operators. In contrast to [37], Satire only computes the first term of the Taylor error expansion. Thus, the computed error bound may not be a sound overestimation. In [1], a sound variation of the abstraction presented in [15], which takes into account also the second-order Taylor term, is presented.

VCFloat [35,3] is a tool that computes rigorous round-off error terms for straight-line Coq expressions. VCFloat does not generate a Coq certificate, instead the computation of the bound is done entirely within Coq. The input program contains a proof template that needs to be instantiated by the user in order to prove the correctness of the computed bounds.

	PRECiSA	FPTaylor	Daisy	VCFloat	Fluctuat	Gappa
proof certificates	✓	✓	✗	✓	✗	✓
conditionals	✓	✗	✗	✗	✓	✗
instability detection	✓	✗	✗	✗	✓	✗
instability analysis	✓	✗	✗	✗	✗	✗
function calls	✓	✗	✗	✗	✓	✗
bounded loops	✓	✗	✗	✗	✓	✗
widening	✓	✗	✗	✗	✓	✗
data collections	✓	✗	✓	✗	✓	✗
rounding modes	✗	✓	✗	✗	✗	✗
fixed-point arith.	✗	✗	✓	✗	✗	✓

Table 2: Comparison of the features of worse-case round-off error analysis tools.

Daisy [13] is a framework for the analysis and optimization of finite-precision computations. It supports both floating-point and fixed-point arithmetic, and it computes estimations for both absolute and relative errors. Daisy does not generate proof certificates, but the external checker FloVer [4] can be used to validate the bounds computed by Daisy. In [25], Daisy has been enhanced with support for arrays and matrices.

Unlike PRECiSA, which targets programs with common constructs such as let-in constructs, conditional, and function calls, FPTaylor, VCFloat, and Daisy are designed to analyze straight-line program expressions. Table 2 summarizes the features of the above-mentioned tools.

Below, PRECiSA 4.0 is compared in terms of accuracy and performance with the following currently maintained open-source tools: Daisy [13] (commit b1705d9), FPTaylor [37] (ver. 0.9.4+dev), and VCFloat2 [3] (commit 10caf1c). This comparison was performed using the standard benchmark suite FPBench [12]. The selected benchmarks involve nonlinear expressions, transcendental functions, and polynomial approximations of functions, taken from equations used in physics, control theory, and biological modeling. These benchmarks and the generated PVS certificates can be found in the PRECiSA distribution. The experimental environment consisted of a 2.6 GHz 6-Core Intel Core i7 with 16 GB of RAM running under MacOS Ventura 13.6.6.

Fig. 7 shows numerical round-off error bounds computed by the aforementioned tools. The default configuration is used for each tool. For PRECiSA, Daisy, and FPTaylor, input variables and constants are assumed to be real numbers. This means that they carry an initial round-off error that has to be taken into consideration in the analysis. VCFloat2 does not support the modeling of the initial rounding, thus the input values are assumed to be perfectly representable as a floating-point. This means that the initial rounding error is not taken into account and it is not propagated. Daisy and FPTaylor use the same round-off error model. However, Daisy relies on data-flow analysis and SMT solvers to compute error bounds, while FPTaylor and PRECiSA use global opti-

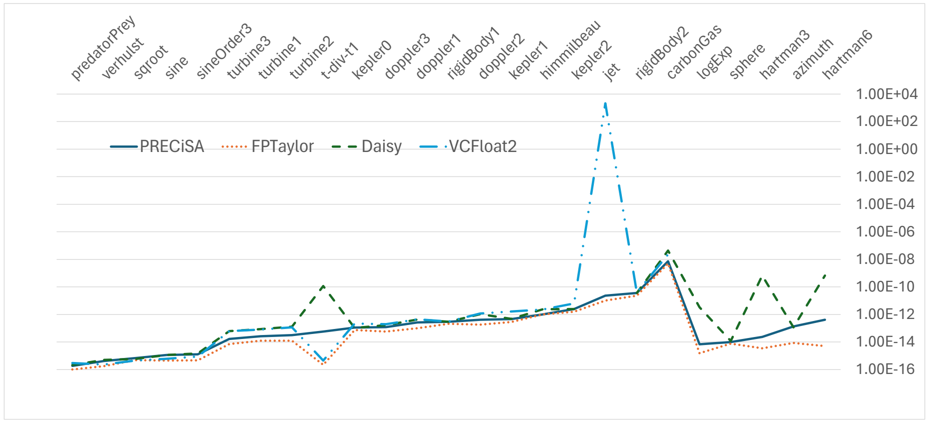


Fig. 7: Experimental results for absolute round-off error bounds.

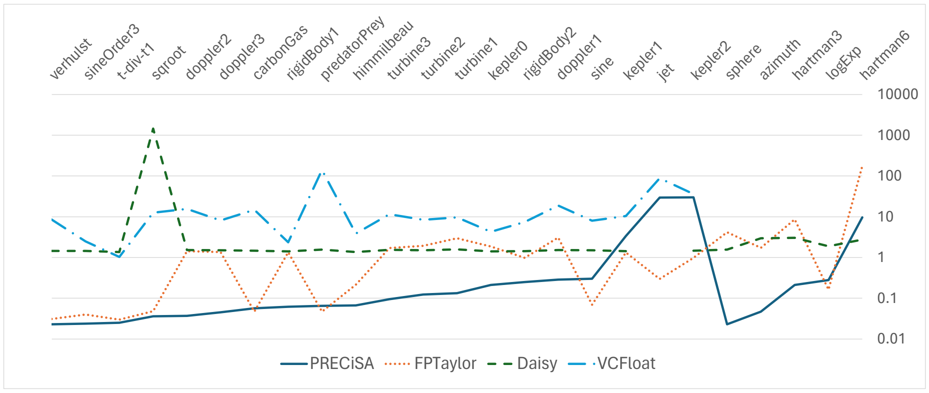


Fig. 8: Times in seconds for the generation of round-off error bounds.

mization methods. The methods used by FPTaylor and PRECiSA are different, but they coincide on certain operations like sum and multiplication. VCFloat uses interval arithmetic with subdivisions, which may be less accurate than the methods used by FPTaylor and PRECiSA. The times for the computation of the bounds are shown in Fig. 8. The performance of PRECiSA is in line with the other similar tools for most of the examples, and for some of the considered benchmarks PRECiSA is the fastest approach. PRECiSA's times also include the generation of the PVS certificates, while Daisy's include the computation of the relative error bound. In summary, for the considered examples, PRECiSA provides a good trade-off between accuracy and performance together with a wide support for arithmetic operations and programming constructs.

Besides worst-case round-off error analysis tools, other tools have been proposed to improve the quality of floating-point software. The static analyzer Astrée [10] automatically detects the presence of potential floating-point run-time exceptions such as overflows and division-by-zero by means of sound floating-point abstract domains [27,7]. *Precision allocation* (or tuning) tools [9,36,14,2] select the lowest floating-point precision for the program variables that is enough to achieve the desired accuracy. *Program optimization* tools [34,43,11,38] improve the accuracy of floating-point programs by rewriting arithmetic expressions in equivalent ones with a lower round-off error. ReFlow [40], initially distributed as part of PRECiSA, automatically extracts floating-point C code from a PVS real number specification. ReFlow implements a code instrumentation that detects unstable conditionals and annotates the code with contracts that relate the floating-point implementation with the real-valued program specification. The annotated code can be used as input to the static analyzer Frama-C [26]. ReFlow relies on PRECiSA to compute the round-off error estimations and the corresponding PVS proof certificates that guarantee their correctness.

8 Conclusion

This paper presents PRECiSA 4.0, the latest release of a NASA open-source static analyzer for floating-point round-off errors. This version of the tool adds several new features and provides support for a wide range of program constructs and mathematical operators. While the majority of other state-of-the-art round-off error analysis tools are limited to straight-line program expressions, PRECiSA targets programs with function calls, predicates, conditionals, and data structures. Conditional instability analysis is particularly challenging to detect and correct by visual code inspection. Issues related to unstable guards have been discovered in NASA libraries implementing geofencing applications [29] and aircraft detect-and-avoid logics [40]. To the best of the authors' knowledge, the conditional instability analysis presented in this work is the first approach that specifically targets the problem of identifying the source of instability in floating-point programs. PRECiSA 4.0 has been used in several applications at NASA, demonstrating its effectiveness and applicability in real-world problems. PRECiSA is at the core of the floating-point C code generator ReFlow , which has been used to generate formally verified floating-point C code for the NASA libraries DAIDALUS [5] and PolyCARP [29].

In the future, the authors plan to add more features to expand even more the applicability of PRECiSA to real-world programs. For instance, support for fixed-point numbers will be added to enable the analysis of quantized neural networks. The symbolic Taylor error expansion introduced in [37] can be integrated into the analysis performed by PRECiSA. These error approximations can be used as an alternative to, or in combination with, the error expressions implemented in PRECiSA. Additionally, the authors plan to enhance the Kodiak tool to support conditional expressions. This feature will improve the accuracy of the round-off error of conditional if-then-else expressions.

Availability. PRECiSA 4.0 is released under NASA Open Source Agreement and it is available at `https://github.com/nasa/PRECiSA`. Additionally, the companion artifact of this submission can be accessed via the following link: `https://doi.org/10.5281/zenodo.12525527`.

References

1. Abbasi, R., Darulova, E.: Modular optimization-based roundoff error analysis of floating-point programs. In: 30th International Symposium on Static Analysis, SAS 2023. Lecture Notes in Computer Science, vol. 14284, pp. 41–64. Springer (2023). https://doi.org/10.1007/978-3-031-44245-2_4
2. Adjé, A., Ben Khalifa, D., Martel, M.: Fast and efficient bit-level precision tuning. In: Proceedings of the 28th International Symposium on Static Analysis, SAS 2021. Lecture Notes in Computer Science, vol. 12913, pp. 1–24. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_1
3. Appel, A.W., Kellison, A.: Vcfloat2: Floating-point error analysis in coq. In: Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024. pp. 14–29. ACM (2024). https://doi.org/10.1145/3636501.3636953
4. Becker, H., Zyuzin, N., Monat, R., Darulova, E., Myreen, M.O., Fox, A.C.J.: A verified certificate checker for finite-precision error bounds in coq and HOL4. In: 2018 Formal Methods in Computer Aided Design, FMCAD 2018. pp. 1–10. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603019
5. Bernardes Fernandes Ferreira, N., Moscato, M.M., Titolo, L., Ayala-Rincón, M.: A provably correct floating-point implementation of well clear avionics concepts. In: Formal Methods in Computer-Aided Design (FMCAD 2023). pp. 237–246. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_32
6. Boldo, S., Muñoz, C.: A high-level formalization of floating-point numbers in PVS. Tech. Rep. CR-2006-214298, NASA (2006)
7. Chen, L., Miné, A., Cousot, P.: A sound floating-point polyhedra abstract domain. In: Proceedings of the 6th Asian Symposium on Programming Languages and Systems, APLAS 2008. Lecture Notes in Computer Science, vol. 5356, pp. 3–18. Springer (2008). https://doi.org/10.1007/978-3-540-89330-1_2
8. Cherubin, S., Agosta, G.: Tools for reduced precision computation: A survey. ACM Computing Surveys **53**(2), 33:1–33:35 (2020). https://doi.org/10.1145/3381039
9. Chiang, W., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous floating-point mixed-precision tuning. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017. pp. 300–315. ACM (2017). https://doi.org/10.1145/3009837.3009846
10. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival: The ASTREÉ Analyzer. In: Proceedings of the 14th European Symposium on Programming (ESOP 2005). Lecture Notes in Computer Science, vol. 3444, pp. 21–30. Springer (2005). https://doi.org/10.1007/978-3-540-31987-0_3
11. Damouche, N., Martel, M.: Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs. 6th Workshop on Automated Formal Methods, AFM 2017 **5**, 63–76 (2017). https://doi.org/10.29007/j2fd
12. 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 Numerical Software Verification, NSV 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10152, pp. 63–77 (2016). https://doi.org/10.1007/978-3-319-54292-8_6

13. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In: 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2018. Lecture Notes in Computer Science, vol. 10805, pp. 270–287. Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_15
14. Darulova, E., Kuncak, V.: Sound compilation of reals. In: Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 235–248. ACM (2014). https://doi.org/10.1145/2535838.2535874
15. Das, A., Briggs, I., Gopalakrishnan, G., Krishnamoorthy, S.: An abstraction-guided approach to scalable and rigorous floating-point error analysis. CoRR **abs/2004.11960** (2020), `https://arxiv.org/abs/2004.11960`
16. de Dinechin, F., Lauter, C., Melquiond, G.: Certifying the floating-point implementation of an elementary function using Gappa. IEEE Trans. on Computers **60**(2), 242–253 (2011). https://doi.org/10.1109/TC.2010.128
17. de Figueiredo, L.H., Stolfi, J.: Affine arithmetic: Concepts and applications. Numerical Algorithms **37**(1-4), 147–158 (2004). https://doi.org/10.1023/B:NUMA.0000049462.70970.b6
18. Ghorbal, K., Goubault, E., Putot, S.: A logical product approach to zonotope intersection. In: Proceedings of the 22nd International Conference on Computer Aided Verification, CAV 2010. Lecture Notes in Computer Science, vol. 6174, pp. 212–226. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_22
19. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: Proceedings of the 13th International Symposium on Static Analysis (SAS 2006). Lecture Notes in Computer Science, vol. 4134, pp. 18–34. Springer (2006). https://doi.org/10.1007/11823230_3
20. Goubault, E., Putot, S.: Perturbed affine arithmetic for invariant computation in numerical program analysis. CoRR **abs/0807.2961** (2008)
21. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Proceedings of VMCAI 2011. Lecture Notes in Computer Science, vol. 6538, pp. 232–247. Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_17
22. Goubault, E., Putot, S.: Robustness analysis of finite precision implementations. In: Proceedings of APLAS 2013. Lecture Notes in Computer Science, vol. 8301, pp. 50–57. Springer (2013). https://doi.org/10.1007/978-3-319-03542-0_4
23. Harrison, J.: HOL light: An overview. In: Proceedings of TPHOLs 2009. Lecture Notes in Computer Science, vol. 5674, pp. 60–66. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_4
24. IEEE: IEEE standard for binary floating-point arithmetic. Tech. rep., Institute of Electrical and Electronics Engineers (2008)
25. Isychev, A., Darulova, E.: Scaling up roundoff analysis of functional data structure programs. In: Proceedings of the 30th International Symposium on Static Analysis, SAS 2023. Lecture Notes in Computer Science, vol. 14284, pp. 371–402. Springer (2023). https://doi.org/10.1007/978-3-031-44245-2_17
26. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing **27**(3), 573–609 (2015). https://doi.org/10.1007/S00165-014-0326-7
27. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: Proceedings of the 13th European Symposium on Programming Languages and Systems, ESOP 2004. Lecture Notes in Computer Science, vol. 2986, pp. 3–17. Springer (2004). https://doi.org/10.1007/978-3-540-24725-8_2

28. Moscato, M., Titolo, L., Dutle, A., Muñoz, C.: Automatic estimation of verified floating-point round-off errors via static analysis. In: Proceedings of the 36th International Conference on Computer Safety, Reliablilty, and Security, SAFECOMP 2017. Springer (2017). https://doi.org/10.1007/978-3-319-66266-4_14
29. Moscato, M., Titolo, L., Feliú, M., Muñoz, C.: Provably correct floating-point implementation of a point-in-polygon algorithm. In: Proceedings of the 23nd International Symposium on Formal Methods, FM 2019. Lecture Notes in Computer Science, vol. 11800, pp. 21–37. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_3
30. Muñoz, C., Narkawicz, A., Hagen, G., Upchurch, J., Dutle, A., Consiglio, M.: DAIDALUS: Detect and Avoid Alerting Logic for Unmanned Systems. In: Proceedings of the 34th Digital Avionics Systems Conference (DASC 2015). Prague, Czech Republic (September 2015)
31. Narkawicz, A., Hagen, G.: Algorithms for collision detection between a point and a moving polygon, with applications to aircraft weather avoidance. In: Proceedings of the AIAA Aviation Conference (2016)
32. Narkawicz, A., Muñoz, C.: A formally verified generic branching algorithm for global optimization. In: Proceedings of the 5th International Conference on Verified Software: Theories, Tools, Experiments, VSTTE 2013. Lecture Notes in Computer Science, vol. 8164, pp. 326–343. Springer (2013). https://doi.org/10.1007/978-3-642-54108-7_17
33. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Proceedings of the 11th International Conference on Automated Deduction, CADE 1992. Lecture Notes in Computer Science, vol. 607, pp. 748–752. Springer (1992). https://doi.org/10.1007/3-540-55602-8_217
34. Panchekha, P., Sanchez-Stern, A., Wilcox, J., Z., T.: Automatically improving accuracy for floating point expressions. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015. pp. 1–11. ACM (2015). https://doi.org/10.1145/2737924.2737959
35. Ramananandro, T., Mountcastle, P., Meister, B., Lethin, R.: A unified Coq framework for verifying C programs with floating-point computations. In: Proceedings of CPP 2016. pp. 15–26. ACM (2016). https://doi.org/10.1145/2854065.2854066
36. Rubio-González, C., Nguyen, C., Nguyen, H., Demmel, J., Kahan, W., Sen, K., Bailey, D., Iancu, C., Hough, D.: Precimonious: tuning assistant for floating-point precision. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13. pp. 27:1–27:12. ACM (2013). https://doi.org/10.1145/2503210.2503296
37. Solovyev, A., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with Symbolic Taylor Expansions. In: Proceedings of the 20th International Symposium on Formal Methods, FM 2015. Lecture Notes in Computer Science, vol. 9109, pp. 532–550. Springer (2015). https://doi.org/10.1007/978-3-319-19249-9_33
38. Thévenoux, L., Langlois, P., Martel, M.: Automatic source-to-source error compensation of floating-point programs. In: 18th IEEE International Conference on Computational Science and Engineering, CSE 2015. pp. 9–16. IEEE Computer Society (2015). https://doi.org/10.1109/CSE.2015.11
39. Titolo, L., Feliú, M., Moscato, M., Muñoz, C.: An abstract interpretation framework for the round-off error analysis of floating-point programs. In: Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2018. Lecture Notes in Computer Science, vol. 10747, pp. 516–537. Springer (2018). https://doi.org/10.1007/978-3-319-73721-8_24

40. Titolo, L., Moscato, M., Feliú, M., Muñoz, C.: Automatic generation of guard-stable floating-point code. In: Proceedings of the 16th International Conference on Integrated Formal Methods (IFM 2020). Lecture Notes in Computer Science, vol. 12546, pp. 141–159. Springer (2020). https://doi.org/10.1007/978-3-030-63461-2_8
41. Titolo, L., Moscato, M., Muñoz, C., Dutle, A., Bobot, F.: A formally verified floating-point implementation of the Compact Position Reporting Algorithm. In: Proceedings of the 22nd International Symposium on Formal Methods, FM 2018. Lecture Notes in Computer Science, vol. 10951, pp. 364–381. Springer (2018). https://doi.org/10.1007/978-3-319-95582-7_22
42. Titolo, L., Muñoz, C., Feliú, M., Moscato, M.: Eliminating unstable tests in floating-point programs. In: Proceedings of the 28th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2018, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11408, pp. 169–183. Springer (2018). https://doi.org/10.1007/978-3-030-13838-7_10
43. Yi, X., Chen, L., Mao, X., Ji, T.: Efficient automated repair of high floating-point errors in numerical libraries. Proc. ACM Programming Languages **3**(POPL), 56:1–56:29 (2019). https://doi.org/10.1145/3290369