VSCode-PRECiSA: a Toolkit for Floating-Point Round-Off Error Analysis
(tool paper)

Paolo Masci\textsuperscript{1}, Laura Titolo\textsuperscript{1}, Marco A. Feliu\textsuperscript{1}, Mariano Moscato\textsuperscript{1}, and Aaron Dutle\textsuperscript{2}

\textsuperscript{1} National Institute of Aerospace\textsuperscript{*}
\{paolo.masci, laura.titolo, marco.feliu, mariano.moscato\}@nianet.org
\textsuperscript{2} NASA Langley Research Center,
\texttt{aaron.m.dutle@nasa.gov}

\textbf{Abstract.} Reasoning about round-off errors in floating-point programs is essential to understand if the computed result is accurate enough with respect to its ideal real number counterpart. VSCode-PRECiSA is a toolkit that integrates the PRECiSA floating-point round-off error analyzer into Visual Studio Code, a mainstream open-source code editor. The toolkit provides an intuitive graphical user interface and offers different analysis views for experimenting and reasoning about how floating-point errors affect the computed result of a program.

1 Introduction

Floating-point numbers are routinely used in hardware and software systems to approximate real numbers. In general, floating-point numbers offer a good compromise between efficiency and precision. However, round-off errors, arising from the difference between a real number and its floating-point representation, accumulate throughout the computation and may result in an unacceptably large divergence between the ideal real number result and the computed floating-point one. In application domains such as avionics, even small round-off errors may have catastrophic consequences when they are not appropriately accounted for. Examples of these errors have been found, for instance, in geofencing applications for UAVs \cite{18} and in aircraft position encoding algorithms \cite{27}.

Several tools have been proposed over the years to reason about floating-point errors (see \cite{2} for an overview). Among these tools, PRECiSA \cite{25} is a static analyzer that correctly estimates the round-off error that may occur in a floating-point program. PRECiSA computes sound and accurate round-off error estimations and it provides support for a large variety of mathematical operators and programming language constructs. PRECiSA, as do the vast majority of state-of-the-art floating-point analyzers, relies on a simple command-line interface for user interaction. This user interface is not suitable for experimenting.

\textsuperscript{*} Research by the first four authors was supported by the National Aeronautics and Space Administration under NASA/NIA Cooperative Agreement NNL09AA00A.
with different analysis inputs and options, and it does not provide a practical environment to locate possible problems and to understand the analysis results.

The contribution of this paper is VSCode-PRECiSA, a toolkit that integrates the PRECiSA static analyzer in Visual Studio Code, a mainstream open-source code editor created by Microsoft and widely used by software developers. VSCode-PRECiSA provides a user-friendly way of performing round-off error analysis. A graphical user interface facilitates the setup and execution of the experiments and interactive plots provide visual aids for improved understanding of analysis results. The toolkit supports different analysis views for reasoning about diverse aspects of floating-point errors: sensitivity analysis, interval analysis, comparative analysis between two different functions, and unstable conditionals analysis to understand if, and for which input values, the control flow of the program is affected by rounding errors.

The rest of the paper is organized as follows. Section 2 provides background information on round-off errors and describes the PRECiSA static analyzer. Section 3 presents the VSCode-PRECiSA functionalities. Section 4 describes the architecture and implementation of VSCode-PRECiSA. Section 5 presents related work. Finally, Section 6 concludes the paper.

2 Background

Floating-point numbers [15] are finite precision representations of real numbers widely used in computer programs. A round-off error originates from the difference between a real number and its floating-point representation. Round-off errors propagate when occurring in arithmetic expressions. The accumulated round-off error is given by a combination of: (i) the propagation of the errors carried by the arguments and (ii) the error introduced by the application of a floating-point operator versus its real number counterpart. Round-off errors in arithmetic expressions may affect the control flow of a program. This happens when the guard of a conditional statement contains a floating-point expression whose round-off error makes the computed Boolean value of the guard differ from the value that would be obtained assuming real arithmetic. Consider the function $\text{sign}(x) = \text{if } x - 0.1 \geq 0 \text{ then } 1 \text{ else } -1$. When the round-off error of $x$ is big enough to alter the evaluation of the Boolean guard, the control flow of the floating-point execution diverges from the real number execution. When this happens, the conditional is said to be unstable. Unstable conditionals cause errors as large as the difference between the results of the different branches. For example, in the above sign function, the error due to unstable conditionals is 2.

PRECiSA [25] is a static analyzer for floating-point programs based on abstract interpretation [5]. PRECiSA accepts as input a floating-point program, which is composed of a set of functions in the language of the Prototype Verification System (PVS) [20], and initial ranges for the input variables of each function. The static analysis generates a sound over-approximation of the floating-point round-off error for each function in the program. In particular, for each program function, PRECiSA computes a symbolic error expression function of the input
variables and their initial errors. Then, the Kodiak global optimizer is used to maximize the symbolic error expression over the input ranges, obtaining a correct numerical overestimation of the floating-point round-off error. To increase the confidence in the computed result, PRECiSA generates proof certificates ensuring that the round-off error estimations are correct. These certificates can be automatically checked in PVS.

The analysis performed by PRECiSA also estimates the error associated with unstable conditionals and computes under which conditions the ideal real numbers path diverges from the floating-point one. These conditions, called instability conditions, are sets of mixed Boolean expressions over reals and floating-point numbers. The Kodiak optimizer paving functionality can be used to compute the regions where these instability conditions are satisfied or to check if the instability conditions are unsatisfiable. The paving algorithm 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 can classify 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, can cause large errors.

PRECiSA provides a command-line user interface. The user invokes the executable `precisa` and passes the program and input files as arguments. Optionally, the user can specify the Kodiak optimization search options max-depths (by default 7) and precision (by default 14) to improve the accuracy or efficiency of the analysis. The computed round-off error estimation and the location of the PVS certificates is displayed in the terminal. A web interface for an older version of PRECiSA is also available but it does not provide any additional functionalities with respect to the basic command-line interface.

3 VSCode-PRECiSA: Functionalities and Examples

VSCode-PRECiSA is a toolkit that integrates the PRECiSA static analyzer in Visual Studio Code. The toolkit provides a graphical user interface with five specialized views, each designed to facilitate the analysis of floating-point round-off errors from a dedicated perspective. The functionalities and utility of the views are explained in the following sub-sections.

3.1 Round-off error analysis

The round-off error analysis view computes a sound estimation of the floating-point round-off error that can occur in a function for given ranges of input variables. The input is a PVS file possibly annotated with comments containing the default input variables ranges (e.g., `@fp-range: nl in [2,59], nz in

\[\text{https://github.com/nasa/PRECiSA/vscode-precisa}\]
The user can explore how the round-off error varies with different input ranges with the help of an editable input form. A plot diagram displays the analysis results using a bar chart. The height of a bar in the chart indicates the value of the estimated round-off error. The chart can contain multiple bars, each representing the analysis results for a different range of input values. The input ranges and value of the round-off error is displayed in a tooltip when the user hovers the mouse over any bar. The tool also shows a results table containing the numerical values of the round-off error for each combination of input ranges and the instability error measuring the divergence of the conditional branches if applicable. The tool provides a feature to save the analysis in the PVS file, by annotating the analyzed function with a tag `@fp-error` followed by information on the analysis input, parameters, and results.

**Example 1.** Fig. 1 shows an example of round-off error analysis in VSCode-PRECiSA for the function `nl_comp`. The function `nl_comp` is part of the Compact Position Reporting (CPR) algorithm, which is responsible for the encoding and decoding of aircraft positions in the Automatic Dependent Surveillance - Broadcast (ADS-B) protocol. CPR subdivides the globe into equally sized zones which are used to locate the position of an aircraft. Given the number of longitude zones (`nl`) and the number of latitude zones (`nz`), the function `nl_comp` computes the lower limit latitude of a longitude zone indexed by `nl`, in the following way.

\[
\text{nl\_comp}(nl, nz) = (180/3.14) \times \arccos \left( \frac{1 - \cos(3.14/(2 \times nz))}{1 - \cos(2 \times 3.14/nl)} \right)
\]

In Fig. 1 different rounding errors are computed for different input ranges. For instance, the first bar is for `nl` ranging in `[2, 59]` and `nz` in `[59, 60]`. In this case, the rounding error is 6.21e–6, while the second bar highlights the results for `nl` ranging in `[2, 30]` that leads to a round-off error of 3.09e–6.
3.2 Interval 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 analyst. Then it computes the floating-point round-off error for each sub-range. This view can be used to gain a better understanding of how the round-off error varies within a given range of input values and it automates some of the manual explorations of input values illustrated in Subsection 3.1. This information can be used, e.g., to re-design the functions to minimize round-off errors in computations.

Example 2. Fig. 2 (left) shows the results for the interval analysis of the \( \text{nl} \_\text{comp} \) function introduced in Example 1. The first bar is the estimated round-off error for the original range, and the other bars are the round-off error for the different sub-ranges. Each group of bars shows the variation with respect to one of the input variables. The first group of bars shows how the round-off error changes as a function of \( \text{nl} \). For smaller values (\( \text{nl} \in [2, 7] \)), the error is approximately \( 1.42 \times 10^{-7} \), while for bigger values (\( \text{nl} \in [52, 59] \)) the error is approximately \( 1.42 \times 10^{-6} \). Note that the error in this function is not sensitive to the changes in \( \text{nz} \) as indicated by the height of the bars in the second group being uniform.

3.3 Sensitivity analysis

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. This view allows an analyst to explore how the round-off error varies when the range of input values changes by a small amount. These types of experiments are useful to detect potentially problematic situations where small variations in the input range limits could lead to substantial changes in the round-off error estimation. Prototypical examples are functions containing \( \text{floor} \) or \( \text{ceiling} \) operators. The endpoints of every input range are perturbed by the uncertainty level and the round-off error analysis is automatically performed for each variation. The overall analysis results are displayed in a bar chart similar to the one used for the interval analysis.

Example 3. Fig. 2 (right) shows the sensitivity analysis results for the function \( \text{nl} \_\text{comp} \) with an uncertainty coefficient of 1\% for the limits of the input ranges.
The first bar is the estimated round-off error for the original range, and the other bars are the round-off error for the different input range variations. It can be noticed that perturbing the input ranges by 1% provokes a variation in the round-off error on the order of $10^{-8}$.

### 3.4 Comparative analysis

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

**Example 4.** Fig. 3 shows the comparative analysis of two different implementations of the $nl\_comp$ function defined in Example 1. A formal study of different finite precision implementations of the CPR algorithm is presented in [11] and [27]. In this study, the following equivalent version of $nl\_comp$ is proposed.

$$nl\_comp2(nl, nz) = \frac{180}{\pi} \cdot \arccos\left(\frac{\sin(3.14/(4 \times nz))}{\sin(3.14/nl)}\right)$$

In Fig. 3, the blue bar depicts the error of the original function which is $6.21e-6$, while the orange bar depicts the error of the simplified version which is $3.37e-14$. The comparative analysis performed by VSCode-PRECiSA, assuming double-precision, reveals that the simplified version should be preferred since it produces a smaller round-off error.
3.5 Conditional instability analysis

The conditional instability analysis view conveys visual information about regions of instability, highlighting which combinations of input variables could alter the control flow of a conditional statement. A 2D-mesh plot is created for a selected pair of variables to produce a diagram akin to a heat-map — red areas in the plot correspond to the regions of possible instability. These regions of instability are computed by PRECiSA with the help of the Kodiak branch-and-bound paving algorithm, as explained in Section 2.

Example 5. Fig. 4 shows the results of the instability analysis for the following function that checks if a point is inside an ellipse-shaped area.

$$\text{pointInEllipse}(x, y) = \text{if } x^2/4 + y^2/9 \leq 10 \text{ then } 1 \text{ else } -1$$

It can be noticed that unstable tests may occur for values close to the border of the ellipse, though regions of instability are not always as obvious (see [18] for an example). Depending on the input ranges, different instability regions are computed by the Kodiak paving. For $x$ and $y$ ranging from $-10$ to 10, an approximation of the border of the ellipse composed of small boxes is displayed. For $x$ and $y$ ranging from 0 to 10, just a quarter of the ellipse border is shown. When $x$ and $y$ are between 0 and 5, no unstable test can possibly occur, thus, VSCode-PRECiSA displays a confirmation message.

4 VSCode-PRECiSA: Architecture and Implementation

Architecture. The toolkit uses a client-server architecture (see Fig. 5) that creates a clear separation of concerns between front-end modules necessary for rendering visual elements and handling interactions with the user, and back-end modules responsible for executing the PRECiSA static analyzer. The standard
Language Server Protocol (LSP) is used to exchange data and commands between the front-end and the back-end.

The front-end uses the WebView framework provided by Visual Studio Code for creating custom user interfaces based on Web Technologies. Multiple front-ends can be opened at the same time, enabling simultaneous analysis of different floating point programs. The front-end includes three main software modules:

- **Client Manager** defines the logic necessary for correct integration of VSCode-PRECiSA in Visual Studio Code, including activation of the extension and handling of settings updates when the user edits them through the Visual Studio Code settings panel.
- **Views Manager** is responsible for handling multiple front-end view instances, including forwarding messages to the active front-end view and resource allocation/deallocation when a front-end view is created/disposed.
- **Toolkit View** defines the logic necessary for rendering the layout of the user interface, including post-processing functions necessary to transform raw textual results communicated by the server back-end into visual plots and tables displayed in the different views. External libraries are used to support the rendering of plots and HTML elements.

The back-end includes three main modules:

- **Server Manager** defines the logic for creating a server instance and establishing the connection with the VSCode-PRECiSA client using the LSP protocol.
- **CodeLens Provider** handles code lenses requests sent by the Visual Studio Code. Code lenses are in-line actionable commands that can be displayed by Visual Studio Code in the document open in the active editor. They are created dynamically based on the content of the document, and represent an alternative to creating menu items or toolbar elements. In VSCode-PRECiSA, two code lenses are used: `estimate-error-bounds`, which is displayed above a function and is used to launch the analysis for that function; and `compare-error-bounds`, which is also displayed above a function and is used to perform a comparison experiment between that function and the one currently loaded in the active VSCode-PRECiSA front-end. These two code lenses are created dynamically when a tag `@fp-function` is introduced above a function definition in the PVS program.
- **Process Worker** embeds the functionalities of the PRECiSA static analyzer in the server back-end. The worker spawns a PRECiSA process in a standard Unix shell on-demand, whenever an analysis request is received from the client. Input parameters passed to the spawned process reflect those specified in the front-end by the user. The output returned by PRECiSA is parsed and transformed into a JSON object that is post-processed by the client.

---

4. [https://microsoft.github.io/language-server-protocol](https://microsoft.github.io/language-server-protocol)
5. [https://code.visualstudio.com/api/extension-guides/webview](https://code.visualstudio.com/api/extension-guides/webview)
Implementation. VSCode-PRECiSA is implemented in TypeScript, a version of the JavaScript language with type annotations. The TypeScript compiler statically checks type correctness of TypeScript programs, and translates TypeScript code into standard JavaScript. The front-end builds on the APIs of Visual Studio Code. Various open-source libraries are used to support rendering of visual elements and create the infrastructure necessary for exchanging data, commands, and events between components, including Plotly.js\(^6\) for rendering charts and Bootstrap.js\(^7\) for setting up the layout and the look & feel of the front-end. The back-end builds on NodeJS, a JavaScript environment for creating Web services, spawning processes, and perform operations on the file system, among other functions. The overall implementation effort to date amounts to 5K lines of code (3.5K for the front-end, and 1.5K for the back-end).

5 Related Work

In recent years, a large number of techniques have been proposed to improve the quality of floating-point code \cite{2}. Worst-case analysis tools provide a sound enclosure of the round-off error that may occur in a program. Different tools use diverse methods to compute this enclosure. For instance, Gappa \cite{10} uses interval arithmetic, Fluctuat \cite{13,14} uses zonotopic abstract domains based on affine arithmetic \cite{12}, Daisy \cite{8} uses an SMT solver-based approach, while FP-Taylor \cite{23}, Real2Float \cite{16}, and PRECiSA all employ different symbolic error expressions in combination with global optimization techniques.

Precision allocation (or tuning) tools, such as FPTuner \cite{3}, Precimomius \cite{22}, Rosa \cite{9}, and POP \cite{1} select the lowest floating-point precision for the program variables that is enough to achieve the desired accuracy. This is essential to constraint the overall round-off error while also maximizing the performance.

Program optimization tools improve the accuracy of floating-point programs by rewriting arithmetic expressions in equivalent ones with a lower round-off error. Examples of these tools are Herbie \cite{21}, AutoRNP \cite{29}, Salsa \cite{6}, and CoHD \cite{24}. Finally, PRECiSA has been recently extended to automatically generate, from a PVS real number specification, a floating-point implementation instrumented to detect when unstable conditional may occur \cite{28,26}.
Except for FPTaylor and the commercial tool Fluctuat, the above-mentioned tools use textual command-line user interfaces which can make the task of experimenting with the analysis burdensome. FPTaylor provides a web interface to compute the error of straight-line expressions. This user interface allows the user to select some options concerning the round-off error model and the optimization search. In contrast with VSCode-PRECiSA, no graphical visualization nor different analysis views are available in FPTaylor.

Fluctuat’s graphical user interface displays values and error ranges for each variable of the program with the help of a bar chart. User-defined domain subdivisions can be specified to obtain a more accurate round-off error analysis. Fluctuat also has the capability of alerting the user when an unstable conditional may occur. However, it does not show a graphical representation of the areas of instability as done in VSCode-PRECiSA. In addition, Fluctuat’s user interface does not provide different analysis views for interval, sensitivity, and comparative analysis.

6 Conclusion

This paper presents VSCode-PRECiSA, a toolkit for the analysis of floating-point programs. This toolkit relies on the PRECiSA [25] static analyzer for computing a correct over-approximation of the round-off error accumulating in a program. VSCode-PRECiSA provides a graphical user interface and five analysis views that help visualizing and understanding how rounding error affects the result of a floating-point computation. In addition, since an integrated development environment for PVS is available for VSCode [17], the PRECiSA toolkit seamlessly integrates with it, making the process of automatically discharging the PRECiSA generated PVS certificate smooth.

In the future, VSCode-PRECiSA will be extended to include the PRECiSA C code generation and instrumentation capabilities presented in [26]. This will support the automatic generation of a floating-point C implementation of a PVS real number specification in a user-friendly environment. The authors plan to provide support for the FPCore input format, which is the standard benchmark language adopted by the FPBench [7] floating-point research community. The FPBench suite includes an exporter that translates FPCore programs to other languages that are input to several analysis, optimization, and tuning tools such as Gappa [10], Salsa [6], FPTuner [3], Herbie [21], or Daisy [8]. This will extend the support of VSCode-PRECiSA to other tools facilitating their comparisons and providing a portfolio of floating-point tools to the user. Finally, the authors plan to extend VSCode-PRECiSA with a suggester that elaborates the information from the analysis and provides tips to the user on how to improve the floating-point code.

References


14. IEEE; IEEE standard for binary floating-point arithmetic. Tech. rep., Institute of Electrical and Electronics Engineers (2008)


