A Step-Function Abstract Domain for Granular Floating-Point Error Analysis

Anthony Dario
University of Oregon
Eugene, Oregon, USA
adario@uoregon.edu

Samuel D. Pollard
Sandia National Laboratories
Livermore, California, USA
spolla@sandia.gov

Abstract

The pitfalls of numerical computations using floating-point numbers are well known. Existing static techniques for floating-point error analysis provide a single upper bound across all potential values for a floating-point variable. We present a new abstract domain for floating-point error analysis which describes error as a function of each variable's value. This domain accurately models the nature of floating-point error as dependent on the magnitude of its operands. We use this domain to effectively handle exceptional values (e.g., NaN), branch instability, and binade boundaries. The granular analysis provides users with a detailed understanding of forward error. We implement the abstract domain in a tool that supports analyzing a subset of C including conditionals, arrays, and arithmetic operators. We compare our implementation with Fluctuat and show how our analysis can improve the error bounds for subranges of possible outputs.

CCS Concepts: • **Theory of computation** → Verification by model checking; • **Mathematics of computing** → **Numerical analysis**; Interval arithmetic; • **Software and its engineering** → Model checking.

Keywords: abstract interpretation, floating point, roundoff error, step functions

ACM Reference Format:
Anthony Dario and Samuel D. Pollard. 2024. A Step-Function Abstract Domain for Granular Floating-Point Error Analysis. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains (NSAD '24), October 22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3689609.3689997

1 Introduction

Floating-point (FP) numbers are a ubiquitous, imperfect representation of the real numbers. This imperfection results in the difference between a numerical computation and its infinite-precision counterpart—its *floating-point error*. This difference can produce real-world consequences [15].

Various static techniques have been developed for analyzing FP error [1, 10, 16, 19, 24, 25]. These techniques provide a sound upper bound on the maximum error a variable can incur during the execution of a program. While useful, a single error bound ignores a key property of FP: a program's error is a complex function of its inputs, but in general, floating-point error depends on the relative magnitude of the operands.

It is often more intuitive to measure relative error[1]. However, this is not without issues. Relative error analysis is typically more challenging to compute and is undefined at 0.

Noting these challenges, we model FP error as a function which at all values overapproximates the true error. We then build an abstract domain [5] by modeling absolute error as a step function. The step function is built by splitting the input domain along every binade, then merging these intervals. Figure 1 shows the effect that crossing binade boundaries has on the error of an approximation of sine as well as our analysis.

With this technique, we gain benefits from both relative and absolute error analysis. This level of granularity allows

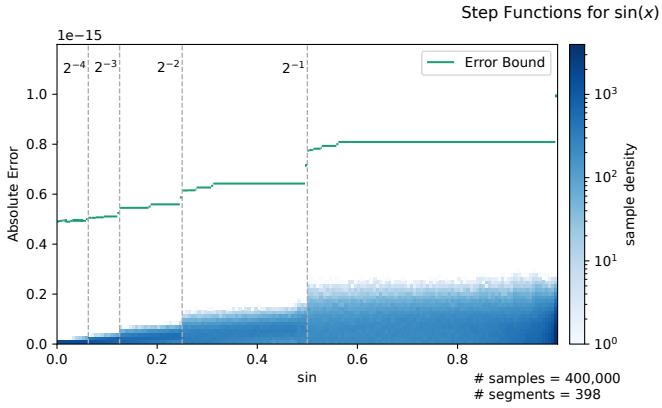

Figure 1. Floating-point error for an approximation of $\sin(x)$ from FPBench [7]. The blue shaded region shows a histogram of the error from randomly-sampled inputs. Binade boundries are vertical lines and green segments show our analysis.

[1]For a real number x and its approximation $\hat{x}$, the relative error is $|x - \hat{x}|/x$ versus absolute error $|x - \hat{x}|$.

NSAD '24, October 22, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1217-3/24/10
https://doi.org/10.1145/3689609.3689997

us to improve error analysis for sub-ranges of an input's domain, allowing the analysis to take advantage of properties which may not hold in all cases (e.g., no divison by zero).

We implement our abstract domain as a static analyzer[2] which supports a subset of C geared towards embedded systems. Our tool supports analysis of elementary FP operations, conditionals, and arrays, and takes as input a C function and acceptable ranges for its FP input variables. For each output, our analyzer builds a step function consisting of worst-case errors partitioned on a set of intervals. These intervals start along binade boundaries, but transform along with the C program's semantics. This approach allows a detailed analysis of a function's behavior and can discover inputs which lead to large error or exceptional behavior. We demonstrate our approach on a set of example problems which closely model the types of numerical software used in safety-critical systems and compare our results with the Fluctuat static analyzer [16].

2 Floating-Point Error

FP error arises from the inability to represent all real numbers in finite precision. Unrepresentable values for a format must be rounded to a representable value. The IEEE 754 standard [26] defines five different rounding schemes. This work assumes the most common: round-to-nearest, ties break to even, but it could be easily extended to other modes.

Roundoff error is often described using *units in the last place* (ulp). For some real number $r \in \mathbb{R}$, $\text{ulp}(r)$ represents the distance between the two FP numbers nearest to r. For round-to-nearest, roundoff error can be bounded with

$$|r - R(F(r))| \leq \frac{1}{2}\,\text{ulp}(r).$$

For floating point numbers $\mathbb{F}$, $R : \mathbb{F} \to \mathbb{R}$ and $F : \mathbb{R} \to \mathbb{F}$ are partial functions which convert between real and FP numbers. In this paper, we apply ulp to floating-point numbers, implicitly converting them to reals. The set of values that are representable with the same (radix-2) exponent (and hence have the same ulp) is called a *binade*.

Branch instability is a class of FP errors that can have a large impact on total error. Branch instability occurs when a branching condition (e.g., an if-statement) evaluates differently between a finite-precision FP context and an infinite precision context. For example, if the branching condition is $x < 0.1$ and $\hat{x} = 0.0995$ in finite-precision, but $x = 0.1$ with infinite-precision, the program will take two different execution paths and cause arbitrarily large error.

We note the true behavior of FP error is complex: some error terms magnify each other, others cancel out, and others still have undefined error (such as underflow). Our work, like others, does not attempt to fully describe the error of FP arithmetic.

[2]available at https://github.com/anthonydario/fp-analysis

$$
\begin{aligned}
A &::= x \mid f \mid i \mid x[A] \mid A \odot A \\
B &::= \mathit{true} \mid \mathit{false} \mid !B \mid A \lessgtr A \\
S &::= x = A \mid x[A] = A \mid S; S \mid \text{if } B \text{ then } S \text{ else } S
\end{aligned}
$$

Figure 2. Syntax of the analyzed subset of C.

$$
\begin{aligned}
&[\![A]\!] : \mathbb{M} \to \mathbb{V} \\
&[\![x]\!]m &&= m\ x \\
&[\![f]\!]m &&= (f, 0) \\
&[\![i]\!]m &&= i \\
&[\![x[A]]\!]m &&= (m\ x)([\![A]\!]m) \\
&[\![A_1 \odot A_2]\!]m &&= [\![A_1]\!]m \odot [\![A_2]\!]m
\end{aligned}
$$

(a) Concrete semantics of arithmetic expressions.

$$
\begin{aligned}
&[\![B]\!] : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M}) \\
&[\![\mathit{true}]\!]M &&= M \\
&[\![\mathit{false}]\!]M &&= \emptyset \\
&[\![!B]\!]M &&= \{m \mid m \in M \wedge m \notin [\![B]\!]M\} \\
&[\![A_1 \lessgtr A_2]\!]M &&= \{m \mid m \in M \wedge [\![A_1]\!]m \lessgtr [\![A_2]\!]m\}
\end{aligned}
$$

(b) Concrete semantics of boolean expressions.

$$
\begin{aligned}
&[\![S]\!] : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M}) \\
&[\![x = A]\!]M &&= \{m[x \mapsto [\![A]\!]m] \mid m \in M\} \\
&[\![x[A_1] = A_2]\!]M &&= \\
&\quad \{m\Big[x \mapsto (m\ x)[[\![A_1]\!]m \mapsto [\![A_2]\!]m]\Big] \mid m \in M\} \\
&[\![S_1; S_2]\!]M &&= [\![S_2]\!]([\![S_1]\!]M) \\
&[\![\text{if } B \text{ then } S_1 \text{ else } S_2]\!]M = \\
&\quad [\![S_1]\!]([\![B]\!]M) \cup [\![S_2]\!]([\![!B]\!]M) \\
&\quad \cup \{m_1[x \mapsto (f_1, |f_1 - f_2| + \epsilon_2)] \mid \\
&\qquad \forall m_1 \in [\![S_1]\!](\mathcal{U}\ M), \forall m_2 \in [\![S_2]\!](\mathcal{U}\ M), \forall x \in m_1, \\
&\qquad (f_1, \epsilon_1) = m_1\ x, (f_2, \epsilon_2) = m_2\ x\} \\
&\quad \cup \{m_1[x \mapsto (f_1, |f_1 - f_2| + \epsilon_2)] \mid \\
&\qquad \forall m_1 \in [\![S_2]\!](\mathcal{U}\ M), \forall m_2 \in [\![S_1]\!](\mathcal{U}\ M), \forall x \in m_1, \\
&\qquad (f_1, \epsilon_1) = m_1\ x, (f_2, \epsilon_2) = m_2\ x\}
\end{aligned}
$$

(c) Concrete semantics of statements.

Figure 3. Concrete semantics.

3 Language and Concrete Semantics

We first define the language under analysis and its concrete semantics. Our concrete semantics captures all possible FP values along with their precise errors.

We analyze a subset of the C language focused on numerical programs shown in Figure 2. The subset includes: variables (x), double-precision FP values (f), integer values (i),

arrays, and elementary arithmetic operators $\odot \in \{+, -, *, /\}$. Integers are included to index arrays. Boolean expressions B consist of *true*, *false*, numerical comparisons (written $\lessgtr \in \{\leq, <, ==, >, \geq\}$), and negation $!B$. Statements S allow for assignment, composition with semicolons, and control flow with if-statements.

The concrete reachability semantics of expressions are given in Figure 3. We model program states as memory, $\mathbb{M} : \mathbb{X} \to \mathbb{V}$, mapping variable names ($\mathbb{X}$) to values ($\mathbb{V}$). Values may be integers, double-precision FP values, or arrays. FP values are modeled as a tuple (f, ϵ) of a FP value, f, and a rounding error ϵ. This work only considers double-precision, but the analysis can easily be extended to other FP formats. We model arrays as functions from integers to numerical values and allow reading and writing from array indices.

The semantics of arithmetic expressions $[\![A]\!] : \mathbb{M} \to \mathbb{V}$ returns the integer or FP value (with error incurred) of the expression given a specific program state.

Boolean expressions $[\![B]\!]$ take as input a set of possible program states and return the subset of states which satisfy the condition. In this way, the semantics of boolean expressions filters memory for the states that satisfy the condition. For *true* and *false* this is all input states and the empty set, respectively. Comparisons ($A_1 \lessgtr A_2$) filter the input set by states that evaluate to true.

Figure 3 shows the concrete reachability semantics of statements $[\![S]\!]$. Assignment ($x = A$ and $x[A_1] = A_2$) updates the left-hand side in every state. We write $m[x \mapsto y]$ to mean the function m updated so that input x produces y. Composition $(S; S)$ gives the possible states after executing the first statement then the second statement.

To correctly capture the effect of branching on error we need to account for stable and unstable executions of either branch. We take the union of the memory produced by both cases. The first two terms filter the memory on the condition and its negation then the appropriate branch is explored. Unstable paths down the "then" and "else" branches are handled by the next two terms. The $\mathcal{U} : \mathcal{P}(M) \to \mathcal{P}(M)$ function filters for memory that would produce branch instability. Then, variables in the memory of the executed branch have their error updated to difference between the executed branch's value and the unexecuted branch's value plus error.

4 Abstract Domain and Semantics

We define an abstract semantics which gives a sound, computable overapproximation of the concrete semantics defined in the previous section. We use *abstract memory* as an abstraction of the set of possible program states from the concrete semantics. Specifically, abstract memory $\mathbb{M}^\sharp : \mathbb{X} \to \mathbb{V}^\sharp$ maps variables to abstract values which overapproximate the values a variable can take in a set of states in the concrete semantics. We describe the domain of these abstract values in § 4.1 and the abstract semantics of the language over abstract values in § 4.2.

4.1 Abstract Domain

We track FP error alongside a variable's possible values in a tuple called a *segment* that consists of an interval (written $x = [x^-; x^+] \in \mathbb{I}$) of possible values and a positive number ($\mathbb{E}$) that overapproximates the error:

$$\mathbb{S} : (\mathbb{I} \times \mathbb{E}).$$

The interval and error bounds are represented by FP numbers. While the bounds may not always be representable, they can be overapproximated by a judicious use of rounding.

To allow us to increase precision for binades closer to 0 we partition $\mathbb{S}$ into multiple segments. The multiple segments describe a piecewise-step function of FP error. We define step functions as sets of segments:

$$\mathbb{P} : \mathcal{P}(\mathbb{S}).$$

Before our analysis computes the abstract semantics of a program, it first converts variables to their abstract value using the abstraction functions $\alpha_{\mathbb{F}} : (\mathbb{F} \times \mathbb{R}) \to \mathbb{P}$ and $\alpha_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{I}$ which map FP constants (and associated error) to single-segment step functions and integers to intervals, respectively:

$$\alpha_{\mathbb{F}}(f, \epsilon) = ([F_\downarrow(f); F_\uparrow(f)], \frac{1}{2}\,\mathrm{ulp}(f)) \tag{1}$$

$$\alpha_{\mathbb{Z}}(i) = [i; i]. \tag{2}$$

For FP numbers we round the written number to the two nearest representable numbers using $F_\downarrow : \mathbb{F} \to \mathbb{F}$ and $F_\uparrow : \mathbb{F} \to \mathbb{F}$ to produce the successor and predecessor of the FP value. This allows us to account for written values that are unrepresentable. Function parameters have unknown values so their range must be specified by the user. Note that (1) bounds the error by 1/2 ulp.

In the abstract domain, arrays are maps of indices to abstract values, $\mathbb{A} : \mathbb{I} \to \mathbb{V}^\sharp$. However, because integers are intervals it is possible the result of an indexing expression is an interval. To index an array with an interval, we take the union of the values from each index in the interval.

Our abstract domain is then

$$\mathbb{V}^\sharp : \mathbb{P} \cup \mathbb{I} \cup \mathbb{A}.$$

4.2 Abstract Semantics

Figure 4 defines the abstract semantics $[\![\cdot]\!]^\sharp$ over the same syntax as Figure 2. We describe the semantics of each syntactic forms in turn.

4.2.1 Arithmetic Expressions. Figure 4a shows the abstract semantics of arithmetic expressions. Accessing variables or arrays involves performing a memory lookup. Constants are mapped to abstract values using the abstraction functions defined in Section 4.1. For arithmetic expressions we define arithmetic operators over abstract values. We use

$$
\begin{aligned}
&[\![A]\!]^\sharp : \mathbb{M}^\sharp \to \mathbb{V}^\sharp \\
&[\![x]\!]^\sharp m^\sharp &&= m^\sharp\ x \\
&[\![f]\!]^\sharp m^\sharp &&= \alpha_{\mathbb{F}}(f) \\
&[\![i]\!]^\sharp m^\sharp &&= \alpha_{\mathbb{Z}}(i) \\
&[\![x[A]]\!]^\sharp m^\sharp &&= (m^\sharp\ x)([\![A]\!]^\sharp m^\sharp) \\
&[\![A_1 \odot A_2]\!]^\sharp m^\sharp &&= [\![A_1]\!]^\sharp m^\sharp \odot_{\mathbb{V}^\sharp} [\![A_2]\!]^\sharp m^\sharp
\end{aligned}
$$

(a) Abstract semantics of arithmetic expressions.

$$
\begin{aligned}
&[\![B]\!]^\sharp : \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
&[\![true]\!]^\sharp m^\sharp &&= M^\sharp \\
&[\![false]\!]^\sharp m^\sharp &&= \bot_{\mathbb{M}^\sharp} \\
&[\![!B]\!]^\sharp M^\sharp &&= m^\sharp[x \mapsto \mathcal{F}_{\neg B}(m^\sharp\ x) \mid x \in m^\sharp] \\
&[\![A_1 \lessdot A_2]\!]^\sharp M^\sharp &&= m^\sharp[x \mapsto \mathcal{F}_{(A_1 \lessdot\ A_2)}(m^\sharp\ x) \mid x \in m^\sharp]
\end{aligned}
$$

(b) Abstract semantics of boolean expressions.

$$
\begin{aligned}
&[\![S]\!]^\sharp : \mathbb{M}^\sharp \to \mathbb{M}^\sharp \\
&[\![x = A]\!]^\sharp m^\sharp &&= m^\sharp[x \mapsto [\![A]\!]^\sharp m^\sharp] \\
&[\![x[A_1] = [A_2]]\!]^\sharp m^\sharp &&= \\
&\quad m^\sharp[x \mapsto (m^\sharp\ x)[[\![A_1]\!]^\sharp m^\sharp \mapsto [\![A_2]\!]^\sharp m^\sharp]] \\
&[\![S_1; S_2]\!]^\sharp m^\sharp &&= [\![S_2]\!]^\sharp([\![S_1]\!]^\sharp m^\sharp) \\
&[\![\text{if } B \text{ then } S_1 \text{ else } S_2]\!]^\sharp m^\sharp = \\
&\quad [\![S_1]\!]^\sharp([\![B]\!]^\sharp m^\sharp) \cup_{\mathbb{M}^\sharp} [\![S_2]\!]^\sharp([\![!B]\!]^\sharp m^\sharp) \\
&\quad \cup_{\mathbb{M}^\sharp} (m^\sharp[x \mapsto \{(i_a, diff(i_a, i_b))\} \mid \\
&\qquad\qquad \forall x, (i_a, e_a) \in [\![S_1]\!]^\sharp(\mathcal{U}_B\ m), \\
&\qquad\qquad (i_b, e_b) \in [\![S_2]\!]^\sharp(\mathcal{U}_B\ m)]) \\
&\quad \cup_{\mathbb{M}^\sharp} (m^\sharp[x \mapsto \{(i_a, diff(i_a, i_b))\} \mid \\
&\qquad\qquad \forall x, (i_a, e_a) \in [\![S_2]\!]^\sharp(\mathcal{U}_B\ m), \\
&\qquad\qquad (i_b, e_b) \in [\![S_1]\!]^\sharp(\mathcal{U}_B\ m)])
\end{aligned}
$$

(c) Abstract semantics of statements.

Figure 4. Abstract Semantics.

standard interval arithmetic [20] for intervals. When mixing intervals and step functions the step function is "cast" to an interval by taking the union of all of its segment's intervals before applying interval arithmetic.

The general idea for arithmetic operators is to apply the operator to each pair of segments from both values and then *merge* any overlapping segments:

$$X \odot_{\mathbb{P}} Y = merge_{\mathbb{P}}(\{x \odot_{\mathbb{S}} y \mid \forall x \in X, y \in Y\}), \tag{3}$$

where $\odot_{\mathbb{P}}$ and $\odot_{\mathbb{S}}$ are arithmetic operators for step functions and segments respectively.

We define segment operators as a combination of interval arithmetic, $\odot_{\mathbb{I}}$, and an error function for propagating error:

$$(i_1, e_1) \odot_{\mathbb{S}} (i_2, e_2) = (i_1 \odot_{\mathbb{I}} i_2, \text{err}_{\odot}(i_1, e_1, i_2, e_2)),$$

Figure 5 shows each operator's error function. Each consists of an error propagation term and resulting rounding

$$
\begin{aligned}
\text{err}_+(i_1, e_1, i_2, e_2) &= e_1 + e_2 + \tfrac{1}{2}\,\text{ulp}(\uparrow i_1 + \uparrow i_2) \\
\text{err}_-(i_1, e_1, i_2, e_2) &= e_1 + e_2 + \tfrac{1}{2}\,\text{ulp}(\uparrow i_1 + \uparrow i_2) \\
\text{err}_*(i_1, e_1, i_2, e_2) &= \uparrow i_1 e_2 + \uparrow i_2 e_1 + e_1 e_2 + \tfrac{1}{2}\,\text{ulp}((\uparrow i_1) * (\uparrow i_2)) \\
\text{err}_/(i_1, e_1, i_2, e_2) &= \tfrac{(\uparrow i_1) e_2 + (\downarrow i_2) e_1}{(\downarrow i_2)^2 - (\downarrow i_2) e_2} + \tfrac{1}{2}\,\text{ulp}(\uparrow i_1 / \downarrow i_2)
\end{aligned}
$$

Figure 5. Operator error propagation.

error term. The functions that propagate existing errors, $\text{err}_{\odot}$, are modified from the analysis in [25] to work on segments.

The $\uparrow : \mathbb{I} \to \mathbb{F}$ and $\downarrow : \mathbb{I} \to \mathbb{F}$ functions select the values in the interval with the largest and smallest magnitude:

$$
\begin{aligned}
\uparrow[l; u] &= \max(|l|, |u|) \\
\downarrow[l; u] &= \begin{cases} 0 & \text{if } l < 0 < u \\ \min(|l|, |u|) & \text{otherwise} \end{cases},
\end{aligned}
$$

To bound the error introduced by rounding, we find a value in the interval, $v \in i_1 \odot_{\mathbb{I}} i_2$, that maximizes[3] $\frac{1}{2}\,\text{ulp}(v)$. It suffices to pick the v with the largest magnitude in the resulting interval $v = \uparrow(i_1 \odot_{\mathbb{I}} i_2)$ as that will maximize the FP's exponent, which in turn maximizes the ulp function. In the case of division, if the interval contains 0 (or small subnormals near 0) then the error is unbounded, so we set the error to $+\infty$.

To bound the error propagated by the error of the operands we pick worst-case values from the operand's intervals to maximize the possible error of the result.

When performing the arithmetic operations on step functions defined in Equation (3), output segments may overlap. For example, if $X = \{([2; 4], e_{x_1}), ([4; 8], e_{x_2})\}$ and $Y = \{([1; 3], e_y)\}$ then $X -_{\mathbb{P}} Y$ will consist of intervals $[-1; 3]$ and $[1; 7]$, overlapping in $[1; 3]$.

To remove this redundancy we perform a $merge_{\mathbb{S}} : \mathbb{S} \times \mathbb{S} \to \mathbb{P}$ operation illustrated in Figure 6. The $merge_{\mathbb{S}}$ operation takes the larger error in any overlapping subintervals:

$$merge_{\mathbb{S}}(s_1, s_2) = \begin{cases} \{s_1\} \cup s_2/s_1 & \text{if } e_1 > e_2 \\ \{s_2\} \cup s_1/s_2 & \text{otherwise} \end{cases}.$$

Here $s_i = (i_i, e_i)$ and $s_2/s_1 : \mathbb{S} \times \mathbb{S} \to \mathcal{P}(\mathbb{S})$ returns all (possibly discontinuous) pieces of s_2 that don't overlap with s_1.

We lift the merge operation to step functions $merge_{\mathbb{P}} : \mathbb{P} \to \mathbb{P}$ by merging all overlapping segments:

$$
\begin{aligned}
merge_{\mathbb{P}}(sf) = \{&merge_{\mathbb{S}}(s_1, s_2) \mid \forall s_1, s_2 \in sf \wedge overlap(s_1, s_2)\} \\
&\cup \{s \in sf \mid \forall s_2 \in sf, s \neq s2 \wedge \neg overlap(s, s_2)\}.
\end{aligned}
$$

The *overlap* predicate is true if the intervals of s_1 and s_2 overlap.

[3] If we were concerned with other rounding modes we could bound the error with $\text{ulp}(n)$ instead.

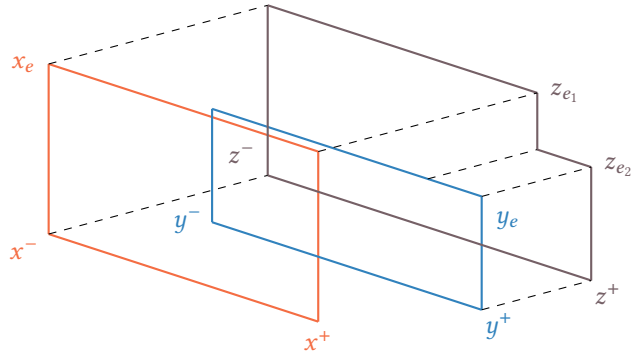

Figure 6. A *merge* operation on segments x and y.

With the arithmetic operators of step functions and the abstraction function defined the abstract semantics are in place for arithmetic expressions.

4.2.2 Boolean Expressions. The abstract semantics of boolean expressions filter the abstract values of a single abstract memory. The values in the abstract memory are restricted to satisfy the conditions. For *true* we return the input memory and for *false* we return the empty memory $\bot_{\mathbb{M}^\sharp}$ which maps every variable to the empty set. For negation, we take the complement of the intervals.

Comparing abstract values involves a filter function $\mathcal{F}_B : \mathbb{P} \to \mathbb{P}$ which removes any segments that cannot satisfy the comparison B. As an example, for $X \leq Y$, X is limited to the set of segments that are less than Y's upper bound, written Y^+, and any segments that cross Y's upper bound are now bounded by Y^+:

$$\mathcal{F}_{(X \leq Y)}(X) = \{s \mid s \in X \wedge s^+ \leq Y^+\} \cup \{([s^-; Y^+], e_s) \mid ([s^-; s^+], e_s) \in X \wedge s^- \leq Y^+ < s^+\}.$$

For negation, we negate the condition we are filtering on. Comparing step functions to constants is similar.

4.2.3 Statements. The abstract semantics of statements modifies the abstract memory. Assignment involves updating the memory to point to the result of the right-hand side expression. Composition is defined similar to the concrete semantics.

The abstract semantics of if-statements account for stable and unstable paths. The first two terms of the semantics of if-statements in Figure 4c compute the error for stable paths by filtering the memory and then exploring either branch. Unstable cases are handled by the last two terms by filtering for potential instabilities determining the error for each variable if either branch was executed.

To detect potential instability, we define the $\mathcal{U}_B : \mathbb{M}^\sharp \to \mathbb{M}^\sharp$ function. The function restricts each segment to only the overlapping portions of their interval:

$$\mathcal{U}_{(X<Y)} m^\sharp = m^\sharp[x \mapsto (x_i \cap [Y^-; Y^+], e_x) \mid \forall (x_i, e_x) \in X].$$

After filtering, we explore both branches and compute the error of each variable by taking the maximum difference in value between the branches:

$$diff([x^-; x^+], [y^-; y^+]) = \max(|x^+ - y^-|, |y^+ - x^-|).$$

We combine the stable and unstable cases by taking the union over abstract memory. The union of abstract memory is the union of all their values:

$$m_1^\sharp \cup_{\mathbb{M}^\sharp} m_2^\sharp = [x \mapsto m_1^\sharp x \cup_{\mathbb{V}^\sharp} m_2^\sharp x \mid \forall x].$$

For the union of abstract values we define the union of two step functions as the union of all their segments merged and the union of (integer) intervals in the standard way for intervals:

$$X \cup_{\mathbb{P}} Y = merge_{\mathbb{P}}(X \cup Y),$$
$$[a^-; a^+] \cup_{\mathbb{I}} [b^-; b^+] = [\min(a^-, b^-); \max(a^+; b^+)].$$

5 Binade Splitting

We can improve our analysis by splitting an operation's result along binade boundaries into segments. As each binade has a different maximum rounding error, smaller binades can be given a more precise error bound.

To accomplish this, we update the arithmetic operators on segments. The idea is to calculate the output interval, then split the interval along all binade boundaries. The resulting subintervals can then be mapped to output segments using error functions slightly modified from the operators defined in Figure 5. First we define the $spl : \mathbb{I} \to \mathcal{P}(\mathbb{I})$ as

$$\begin{aligned} &spl([a^-; a^+]) = \\ &\quad \{[2^n; 2^{n+1} - \text{ulp}(2^{n+1})] \mid n \in \mathbb{N} \wedge a^- < 2^n \wedge a^+ > 2^n + 1\} \\ &\cup \{[a^-; 2^n - \text{ulp}(2^n)] \mid n = \text{fexp}(a^-)\} \\ &\cup \{[2^n; a^+] \mid n = \text{fexp}(a^+)\}. \end{aligned}$$

Here, $\text{fexp}(f) : \mathbb{F} \to \mathbb{Z}$ gives the exponent of the FP value f and $n \in \mathbb{Z}$. The first term finds all subintervals that span an entire binade. The second and third terms are intervals formed from the lower and upper bounds of input interval and their nearest binade boundary.

To propagate error to these split intervals we slightly modify the error operators in Figure 5. The propagation component is unchanged but the rounding error is improved by taking the upper bound of the output interval, i_o:

$$\begin{aligned} \text{err}_+(i_1, e_1, i_2, e_2, i_o) &= e_1 + e_2 + \frac{1}{2}\,\text{ulp}(\uparrow i_o) \\ \text{err}_-(i_1, e_1, i_2, e_2, i_o) &= |e_1 + e_2| + \frac{1}{2}\,\text{ulp}(\uparrow i_o) \\ \text{err}_*(i_1, e_2, i_2, e_2, i_o) &= \uparrow i_1 e_2 + \uparrow i_2 e_1 + e_1 e_2 + \frac{1}{2}\,\text{ulp}(\uparrow i_o) \\ \text{err}_/(i_1, e_2, i_2, e_1, i_o) &= \frac{(\uparrow i_1) e_2 + (\downarrow i_2) e_1}{(\downarrow i_2)^2 - (\downarrow i_2) e_2} + \frac{1}{2}\,\text{ulp}(\uparrow i_o). \end{aligned}$$

```
x = {[0; 1.57079632679489656], 0.0}
```

(a) The specification file - `sine.spec`

```
double sine_taylor(double x) {
    return x - (x*x*x) / 6.0 +
           (x*x*x*x*x) / 120.0 -
           (x*x*x*x*x*x*x) / 5040.0; }
```

(b) The code file - `sine.c`

```
var,type,low,high,err
retval,flt,1.7192e-2,2.6568e-2,4.8992e-16
...
retval,flt,1.0619e+0,1.1465e+0,1.1421e-15
```

(c) The output file - `sine.csv`.
Numbers have been truncated to four significant digits

Figure 7. The input and output files when analyzing $\sin(x)$

We update the arithmetic operators on segments to take into account this splitting:

$$(i_1, e_1) \odot (i_2, e_2) = \{(i, err_{\odot}(i_1, e_1, i_2, e_2, i)) \mid i \in spl(i_1 \odot i_2)\}.$$

We use these updated segment arithmetic operators in Equation (3) to provide an improved bound on the error.

6 Implementation

We implement our step function abstract domain in OCaml. Our analyzer accepts a C source file, the name of a function to analyze, and a specification file containing preconditions for function input. Our analyzer then determines the range of values and the error for each variable declared in the specification file. Figure 7 shows the input and output files from an analysis of the sine approximation from Figure 1.

Specification files are written by the user to provide bounds on the function parameters' value and error. These files contain variable names and associated segments. Segments are written as `([lb ; ub], err)` with `lb` and `ub` being the variable's lower and upper bound and `err` being the error's upper bound. Multiple segments can be included for a single variable. We show an example specification file in Figure 7a where the input ranges from 0 to $\pi/2$ with no error.

Our tool ingests C code using the CIL intermediate representation (IR) of C in OCaml, which is updated and maintained by the Goblint static analysis project [23]. We then transform the IR to the internal representation described in Section 4. We chose CIL as it is a mature project that has been used in other static verification tools such as Frama-C and Goblint. A program's FP variables are then abstracted to using the functions described in Equations (1) and (2). To use FP rounding modes in OCaml we defined C functions that allow for the specification of the rounding mode and linked them using OCaml's foreign function interface.

Figure 7c shows the output file of an analysis. The output is a CSV where each row is a segment containing the lower bound (`low`), upper bound (`high`), and error (`err`). The `var` column specifies which variable in memory the segment belongs to. The `type` column specifies if the variable is a FP value or an integer.

7 Evaluation

We have tested our analysis on a selection of benchmarks from FPBench [7]. The most direct comparison is Fluctuat [16], because both analyze C code directly using abstract interpretation, but these benchmarks are often compared in other tools in the field [8, 9, 24].

Because of the detailed nature of our analysis, it is difficult to directly compare with other tools. In general, our analyzer's overall upper bound on the error is worse, however the analysis shows an improvement on error bound for certain output values. Moreover, several other analyzers do not support conditionals or arrays.

Figure 8 shows analysis for five different FPBench functions (cav10 is plotted twice for reasons we will explain). In all benchmarks, the y-axes show absolute error as the difference between 64-bit double-precision FP and 200-bit MPFR float results [14].

The x-axes show the output of the functions. The benchmarks rigidBody1 and doppler1 are ternary and the rest are unary. The blue density map shows the observed error of the output. We randomly sample 1 million inputs, but do not attempt to select inputs that maximize observed error.

The green lines show the segments of our analyzer and and the orange lines shows the error produced by Fluctuat. Generally, Fluctuat provides tighter bounds in the general case but our analysis shows tighter bounds for certain outputs, which can help guide further analysis and verification.

We point out in the rigidBody benchmark, there is a single triple of inputs which has a larger observed error than the Fluctuat bound. This is not a plotting artifact; we invesigated this but were not able to determine why Fluctuat returns an unsound bound in this case.

We plot the cav10 benchmark twice to show Fluctuat's two modes: one which supports branch instability, but has similar imprecision as ours (absolute error of about 100), and another, more precise but unsound with respect to branch instability. Our analysis correctly predicts areas of unbounded error (not plotted, but returned as $+\infty$ in our tool).

Table 1 shows our runtime results on the benchmarks plotted in Figure 8 using a Macbook M1. Fluctuat scales better for analysis of larger programs. We discuss potential optimizations for our approach in § 9. A more in-depth survey of tools and timing results is provided by Solovyev et al. [24].

The lack of precision for larger values is expected. The step function abstract domain uses intervals to calculate the range of values for a variable , which are less precise than

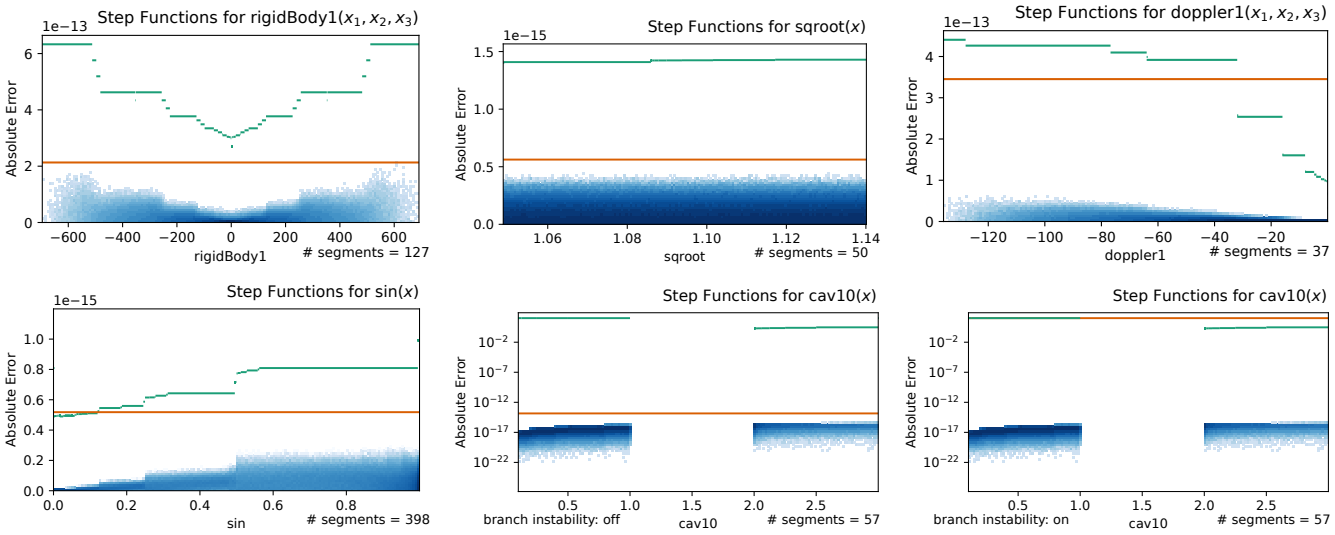

Figure 8. Results of several benchmarks from FPBench compared with Fluctuat and 1 million randomly-sampled inputs.

Table 1. Timing results for Fluctuat versus our tool.

Benchmark	Fluctuat	Step Functions
rigidBody1	.01s	.43s
sqroot	.01s	.33s
doppler1	.02s	.03s
sin	.02s	.09s
cav10	.02s	.23s

Fluctuat's model of zonotopes and so produces larger error bounds. However, this approach demonstrates the practicality of the step function domain and we mention opportunities for improving the precision of our analysis in § 9.

8 Related Work

Several tools for static FP error analysis have been developed. PRECiSA [25], Astrée [6], and Fluctuat [16] are based on abstract interpretation. These tools provide a single bound for error analysis but use sophisticated abstract domains to tighten their overapproximations. Both Fluctuat and Astrée provide better approximations compared to interval arithmetic, but still rely on linear relationships between variables which can perform poorly on non-linear operators [20, 21]. Other abstract domains have been formulated which can better handle loops and more complex arithmetic [4, 12], but since Astrée and Fluctuat are closed source it is difficult to examine which are used in practice.

Other automated-reasoning tools for FP error analysis include Daisy [8], FPTaylor [24], and Gappa [11], which each support different programming language constructs. FPTaylor and Gappa require straight-line code, PRECiSA and Daisy support conditionals. Each require expressing programs in domain-specific languages, though some have been used in pipelines full-program analysis. Since we analyze C code directly, our most direct comparison is Fluctuat.

Proof assistants allow users to build constructive, machine-checkable proofs of bounds on their FP error using functional models of their code. These formalizations have been developed in Coq [2] and PVS [3]. Defining and proving these functional models is a manual process that can prove difficult and requires skilled analysts, though VCFloat2 and PVS do provide some automation capabilities [1].

Recent work extending PRECiSA, called ReFlow, uses abstract interpretation, code generation, proof assistants, and annotation languages to build correct-by-construction implementations of FP programs in C [13]. This approach generates verification conditions in Frama-C [18] which are then dispatched to automated or interactive provers.

In contrast, the goal of our tool is to provide a fully-automated, detailed understanding of FP errors which then could facilitate more formal proofs using Coq or PVS if the effort is deemed necessary.

9 Future Work and Conclusion

There are some clear directions to continue developing our step function abstract domain: our underlying value approximation, interval arithmetic, is known to be imprecise and function calls and loops are absent from our analysis.

More sophisticated abstract domains, such as zonotopic (used in Fluctuat [16]) and octagon (used in Astreé [6]) would improve error analysis but require splitting the error function over a more complex value space.

Adding support for function calls would allow analysis of many more programs. Function calls introduce new challenges as they complicate the control flow of the program. Multiple techniques for supporting functions have been developed and provide possible extensions to our tool [17, 22].

As noted in § 7, our technique as implemented may not scale well to larger programs. This could be improved by bounding the total number of segments the analyzer creates; this would limit the complexity per floating-point operation but may reduce precision.

Handling unbounded loops is tricky for any static analysis. In abstract interpretation, the typical approach is to find a fixpoint of iterations of the loop body. One approach is using a widening operator, but in practice, the results are often too coarse to be useful, so analysts instead bound loop iterations. Extending our analysis with bounded loops is straightforward, while defining a precise widening operator remains a more difficult goal.

In this paper, we presented a new step function abstract domain for analyzing floating-point error. The domain provides a more granular error analysis for floating-point code by mimicking the behavior of floating-point error. We implemented the domain and performed comparisons with Fluctuat. Evaluations show that the granular analysis can improve error bounds for subsets of variable outputs.

Acknowledgment

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

[1] Andrew Appel and Ariel Kellison. 2024. VCFloat2: Floating-Point Error Analysis in Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs* (London, UK) *(CPP)*. Association for Computing Machinery, New York, NY, USA, 14–29.

[2] Sylvie Boldo and Guillaume Melquiond. 2017. *Computer Arithmetic and Formal Proofs: Verifying Floating-Point Algorithms with the Coq System* (1st ed.). ISTE Press - Elsevier, United Kingdom. 326 pages.

[3] Sylvie Boldo and César Mu noz. 2006. *A High-Level Formalization of Floating-Point Numbers in PVS*. Technical Report 2006-214298. National Institute of Aerospace. https://shemesh.larc.nasa.gov/fm/papers/Boldo-CR-2006-214298-Floating-Point.pdf.

[4] Liqian Chen, Antoine Miné, and Patrick Cousot. 2008. A Sound Floating-Point Polyhedra Abstract Domain. In *Programming Languages and Systems*. Springer, Berlin, Heidelberg, 3–18.

[5] Patrick Cousot. 2021. *Principles of Abstract Interpretation*. MIT Press.

[6] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTRÉE analyzer. In *Programming Languages and Systems: 14th European Symposium on Programming (ESOP)*. Springer, 21–30.

[7] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Jason Qiu, Alex Sanchez-Stern, and Zachary Tatlock. 2016. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In *International Workshop on Numerical Software Verification (NSV)*.

[8] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer International Publishing, Thessaloniki, Greece, 270–287.

[9] Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 8 (mar 2017), 28 pages.

[10] Florent De Dinechin, Christoph Lauter, and Guillaume Melquiond. 2010. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Trans. Comput.* 60, 2 (2010), 242–253.

[11] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. 2011. Certifying the Floating-Point Implementation of an Elementary Function Using Gappa. *IEEE Trans. Comput.* 60, 2 (Feb. 2011), 242–253.

[12] Jérôme Feret. 2005. The arithmetic-geometric progression abstract domain. In *Verification, Model Checking, and Abstract Interpretation: 6th International Conference, Paris, France. (VMCAI)*. Springer, 42–58.

[13] Nikson Bernardes Fernandes Ferreira, Mariano M. Moscato, Laura Titolo, and Mauricio Ayala-Rincón. 2023. A Provably Correct Floating-Point Implementation of Well Clear Avionics Concepts. In *Formal Methods in Computer-Aided Design (FMCAD)*. 237–246. https://ntrs.nasa.gov/citations/20230007436

[14] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.* 33, 2 (June 2007), 1–15.

[15] Ivan Fratric. 2019. The Curious Case of Convexity Confusion. https://googleprojectzero.blogspot.com/2019/02/the-curious-case-of-convexity-confusion.html.

[16] Eric Goubault and Sylvie Putot. 2011. Static analysis of finite precision computations. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 232–247.

[17] Daniel Kästner and Christian Ferdinand. 2014. Proving the absence of stack overflows. In *Computer Safety, Reliability, and Security: 33rd International Conference, Florence, Italy, September 10-12, 2014. Proceedings 33 (SAFECOMP)*. Springer, 202–213.

[18] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (May 2015), 573–609.

[19] Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2017. On Automatically Proving the Correctness of Math.h Implementations. *Proc. ACM Program. Lang.* 2, POPL, Article 47 (dec 2017), 32 pages.

[20] Antoine Miné. 2004. Relational abstract domains for the detection of floating-point run-time errors. In *European Symposium on Programming*. Springer, 3–17.

[21] Antoine Miné. 2006. The octagon abstract domain. *Higher-order and symbolic computation* 19 (2006), 31–100.

[22] Corneliu Popeea and Wei-Ngan Chin. 2006. Inferring disjunctive postconditions. In *Annual Asian Computing Science Conference*. Springer, 331–345.

[23] Michael Schwarz, Simmo Saan, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. 2021. Improving Thread-Modular Abstract Interpretation. In *Static Analysis (SAS)*. Springer International Publishing, Cham, 359–383.

[24] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 2 (dec 2018), 39 pages.

[25] Laura Titolo, Marco A Feliú, Mariano Moscato, and César A Muñoz. 2018. An abstract interpretation framework for the round-off error analysis of floating-point programs. In *Verification, Model Checking, and Abstract Interpretation: 19th International Conference, Los Angeles, CA, USA (VMCAI)*. Springer, 516–537.

[26] Working Group for Floating-Point Arithmetic. 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84.

Received 2024-07-10; accepted 2024-08-18