Robustness Analysis of Loop-Free Floating-Point Programs via Symbolic Automatic Differentiation

Arnab Das*, Tanmay Tirpankar†, Ganesh Gopalakrishnan‡
School of Computing
University of Utah
Salt Lake City, Utah, USA
Email:*arnab.d.88@gmail.com, †tirpankartanmay@gmail.com, ‡ganesh@cs.utah.edu

Sriram Krishnamoorthy
Google[1]

California, USA

***Abstract*—Automated techniques for analyzing floating-point code for roundoff error as well as control-flow instability are of growing importance. It is important to compute rigorous estimates of roundoff error, as well as determine the extent of control-flow instability due to roundoff error flowing into conditional statements. Currently available analysis techniques are either non-rigorous or do not produce tight roundoff error bounds in many practical situations. Our approach embodied in a new tool called SEESAW employs *symbolic reverse-mode automatic differentiation*, smoothly handling conditionals, and offering tight error bounds. Key steps in SEESAW include weakening conditionals to accommodate roundoff error, computing a symbolic error function that depends on program paths taken, and optimizing this function whose domain may be non-rectangular by paving it with a rectangle-based cover. Our benchmarks cover many practical examples for which such rigorous analysis has hitherto not been applied, or has yielded inferior results.**

***Index Terms*—Floating Point Numbers, Rounding, Symbolic Automatic Differentiation, Computational Graphs, Optimization**

I. Introduction

Floating-point arithmetic is fundamental to many areas of computing including HPC and machine learning, geometric algorithms, and embedded systems. Given that floating-point code approximates real numbers, the consequences of this approximation must be rigorously analyzed to be within acceptable margins. Two of these consequences are studied in this paper: (1) how much rounding error is introduced, and (2) are the control flow paths different when a program is studied under the real number model versus its floating-point model. The latter is popularly known as *control-flow instability*. In this paper, we study these two *non-robustness* aspects of *loop-free* floating-point programs. In other words, the codes analyzed in our work involve a sequence of assignments and branches, but not loops. While loop-free programs may strike as a serious limitation, in reality algorithms found within geometric libraries such as computing the tightest sphere enclosing a cloud of points, the closest point to a line, etc., are of this nature. There is no point entertaining the added complexity of loops, given that today's solutions are inadequate even within the loop-free class.

The simple example in Figure 1 of a program over two floating-point inputs `x` and `y` in intervals $(0.1, 1.0)$ brings out the kinds of non-robustness we are after. Imagine running this program by modeling it under single-precision and later under double-precision: are there inputs for which the control flow paths differ? The answer is yes: by sampling the input space to a fine resolution (purely for demonstration), we can determine that by feeding $x = 9.8153645431765959$ and $y = 9.4153646206626345$, the truth-value of the program control-flow predicate P_1 changes (also, on either side of these x and y values, the control flows with respect to P_1 agree). Further, observe that the assignment to g differs in these paths under P_1, meaning that the final result res changes in response to this control-flow change. We call this the *instability jump* of the output. Determining all control-flow instabilities and the consequent instability jumps clearly requires roundoff error analysis (it is the rounding error flowing into the conditionals that causes instabilities).

While we shall present related work in detail later, it is worth quickly mentioning that three main related efforts address these correctness issues and offer *rigorous* solutions—meaning, solutions across *intervals* of inputs (not just samplings of input points). Darulova [1] presents a formal approach to characterize non-robustness of this sort. Ghorbal [2] introduces constrained affine sets over their Zonotopic abstract domains to handle constraints—an approach that, to our best knowledge also underlies the closed-source tool FLUCTUAT [3] (available for experimentation under license). Finally, the PRECISA [4] tool handles conditionals through a denotational semantics-based approach.

In comparison, we contribute a rigorous *symbolic* method of error analysis embodied in a new tool called SEESAW. Seesaw builds on the core technology of our recent tool SATIRE [5] (namely, symbolic reverse-mode automatic differentiation or

[1]Work done while at Pacific Northwest National Laboratory
Supported in part by NSF CISE Grants 1704715, 1917141 and 1918497

```
1   inputs { x fl64 : (0.1, 1.0);
2            y fl64 : (0.1, 1.0); }
3   outputs { res; }
4  // Program definition
5   exprs {
6    h = (y/x) + x ;
7    g = x + x*y ;
8    if ( x − y < 0.4 ) then // P1
9     g = 1 + 1/g ;
10   else
11    if ( (x*x > 0.25) && (y*h <= x*x) )
            then // P2
12     g = h + x*y ;
13    endif
14   endif
15   res = g + y ; }
```

Fig. 1: For this running example, its output **res** is plotted in Figure 2 as follows. The top plot shows x and y over the input intervals $(0, 10)$. The zoomed versions show the function behaviour at the lower and higher ends of the input interval (notice the instability jumps shown).

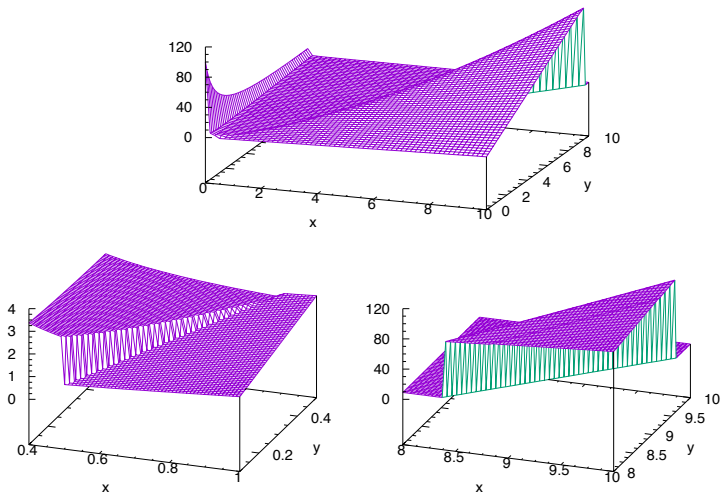

Fig. 2: Function plot of output **res**; Top: $x, y \in [0.01, 10.0]$; Bottom-left: $x \in [0.4, 1.0], y \in [0.1, 0.5]$; Bottom-right:$x, y \in [8.0, 10.0]$

"AD") that can scale error analysis to expressions with over 2 million operator nodes. Key contributions that SEESAW makes over SATIRE are an extension of AD to handle conditionals, calculating instability jumps, handling the global optimization problem over non-rectangular domains through "paving" (all to be elaborated later).

For codes without any conditionals, SEESAW offers the same level of scalability as SATIRE. For codes with a few conditionals, this level of scalability is retained. For codes with many conditionals, SEESAW will suffer from path explosion. Fortunately, many practical programs contain largely conditional-free portions and a few "top level" governing conditionals.

Comparable open-source rigorous tools do not attain this level of scalability as well as the tightness of error estimates. When applied to the example in Figure 1, PRECISA reports an instability jump of 1003 with a rounding error bound of 8.72e-09. FLUCTUAT only reports the existence of instability, without giving a magnitude for it. FLUCTUAT reports a rounding error bound of 3.22e-10 over the entire domain. FLUCTUAT reports the existence of instability without calibrating instability jump magnitudes. SEESAW, on the other hand, reports the instability jump to be of magnitude 105.9 and the rounding-error bound to be 5.5e-14. Overall, across our benchmarks, SEESAW has obtained tighter bounds than PRECISA and FLUCTUAT. The relative tightness of roundoff error bounds estimated by SEESAW has been confirmed through shadow value calculations.

In terms of novelty, SEESAW accommodates conditional programs smoothly in a symbolic AD framework built using Python and SymEngine [6]. SEESAW handles optimization over non-rectangular constraint regions by *paving* the domain with a collection of minimally overlapping rectangular regions using the off-the-shelf tool Realpaver [7], allowing the use of efficient rectangular domain solvers.

Key Contributions:

- SEESAW provides rigorous estimates of the roundoff errors on outputs as well as instability jump values. It also ranks the instability bounds, helping the designer address the most significant ones first.
- SEESAW includes a statistical analysis method that involves sampling the symbolic error expression generated. This approach helps pragmatically calibrate the severity of worst-case error bounds.
- We also contribute a benchmark suite of loop-free programs to evaluate future tools in this space (such benchmarks were unavailable before our work), and also release the tool (https://github.com/arnabd88/Seesaw.git).

Roadmap: §II provides relevant background. §III provides a complete walk-through through our running example. §IV provides all the rigorous steps involved in our symbolic reverse-mode AD, computing error expressions, and calculating instability jumps. §V provides our evaluation results. §VI provides discussions and conclusions.

II. BACKGROUND

Consider a sequential straight-line floating-point expression of m input variables $\mathbf{x} = \{x_1, \cdots, x_m\}$ executing n floating-point operations. We present the program in a single static assignment form $s_i = f_i(x_1, \ldots, x_m, s_1, s_2, \ldots, s_{i-1})$. Here, f_i denotes the arithmetic operation being executed in the i^{th} right-hand side, where f_i is an arbitrary arithmetic operation or elementary function. The final output is s_n. Let $\tilde{f}_i$ denote the finite precision approximation of f_i. Application of $\tilde{f}_i$

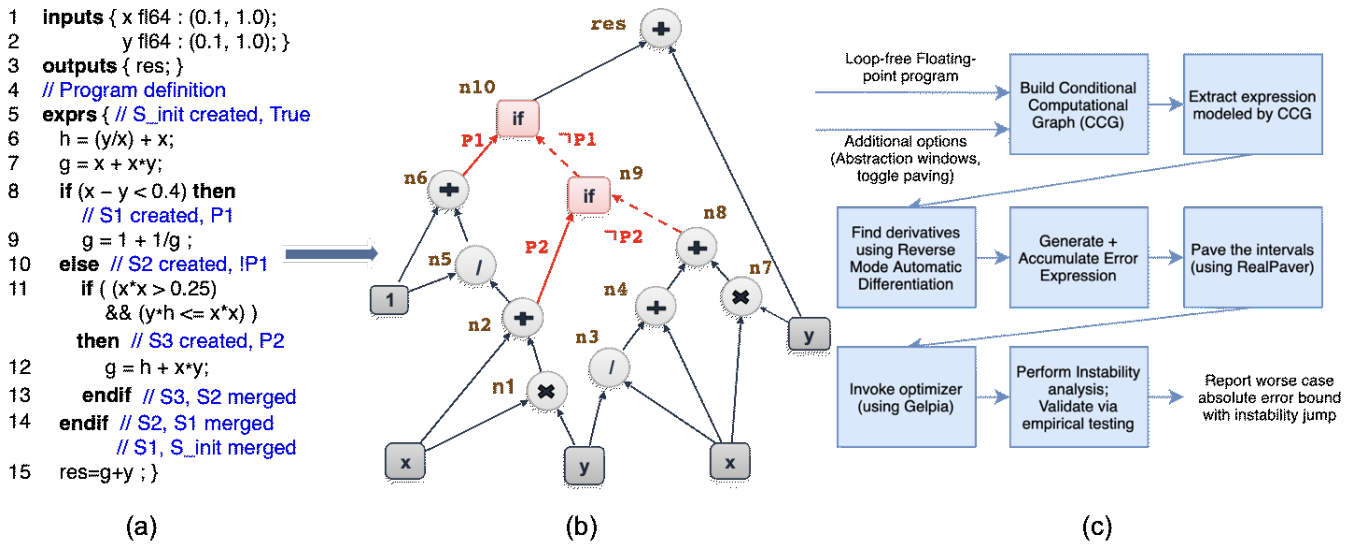

Fig. 3: (a) Example Program with Scopes Illustrated; (b) Translated Conditional Computational Graph (CCG); (c) Overview of Seesaw

will encounter a rounding error (e_{s_i}) to produce $\tilde{s_i}$, such that $\tilde{s}_i = s_i + e_{s_i}$. Additionally, input variables may also contain rounding error from previous stages. We model this by changing $\mathbf{x}$ to $\tilde{\mathbf{x}}$ where $\tilde{\mathbf{x}} = \{x_1 + e_{x_1}, \ldots, x_m + e_{x_m}\}$. δ_i is the floating-point round-off error generated at node i. The absolute value of δ_i is bounded by the *unit roundoff* $\mathbf{u}$ (or machine epsilon) value [8].[1]Based on first-order error analysis, one can write the error accumulated at the output, e_{s_n}, as

$$e_{s_n} = s_n \cdot \delta_n + \sum_{j=1}^{n-1} (\partial s_n / \partial s_j) \cdot e_{s_j} + \sum_{j=1}^{m} (\partial s_n / \partial x_j) \cdot e_{x_j} \quad (1)$$

Here, the term $s_n \cdot \delta_n$ models the error generated at the output node s_n. This error is calculated in terms of the *local error* e_{s_j} generated at intermediate nodes (which, in turn, are calculated using the very same first-order analysis captured by Equation 1). We also include the effect of the input errors, summed over the m input nodes. Observe that $s_n = f_n(x_1, \ldots, x_m, s_1, s_2, \ldots, s_{n-1})$. Thus, we must obtain $(\partial s_n / \partial s_j)$ where j is summed over $1 \ldots n-1$. This summation is done using reverse-mode automatic differentiation (AD) which uses the chain rule. The primary goal of our error analysis is to obtain a tight bound on the worst-case output error – specifically, obtain the suprema for the derived error expression in (1) across intervals of input values.

Reverse Mode AD: Let an n-output computation over m inputs be modeled via function S, where $S : \mathbb{R}^m \to \mathbb{R}^n$. Furthermore, let $S(\mathbf{x}) = (h \circ g)(\mathbf{x})$ capture a decomposition of S via functions h and g (modeling how computational DAGs are recursively processed) where $g : \mathbb{R}^m \to \mathbb{R}^k$ and $h : \mathbb{R}^k \to \mathbb{R}^n$. Then, the error in $S(\mathbf{x})$ can be computed via reverse-mode AD by computing a Jacobian, J, where the J_{ij} component is

$$J_{ij} = \frac{\partial S_i}{\partial x_j} = \frac{\partial h_i}{\partial g_1}\frac{\partial g_1}{\partial x_j} + \cdots + \frac{\partial h_i}{\partial g_k}\frac{\partial g_k}{\partial x_j} \quad (2)$$

In SEESAW, we extend the efficient dynamic-programming implementation of this approach introduced in [5] to include conditional paths, as illustrated in §III and detailed in §IV.

III. A Complete Walkthrough

Figure 3(c) shows the workflow of SEESAW. The input to SEESAW is a straight-line program with conditionals given by the command syntax in Figure 4. The whole program to be analyzed, $pgm(Vars;\ C;\ out)$, consists of variable declarations $Vars$, a command C, and a single output out (all outputs handled similarly). The rest of the syntactic categories are familiar (we assume that negation has been pushed innermost). SEESAW first obtains a *conditional computational graph* (CCG) as illustrated in Figure 3(b). We extract a computational expression from the CCG and subject it to reverse-mode AD as will be defined precisely in §IV. We now walk through each of these phases.

```
1  E ::= op(E1) | op(E1,E2) | x ∈ Vars | c ∈ Const;
2  BE ::= E1 rel E2 | BE1 bop BE2;
3  C ::= x := E | C1;C2 | if BE then C endif
4        | if BE then C1 else C2 endif | skip;
5  P ::= pgm(Vars; C; out);
```

Fig. 4: Syntax of Input Programs

Translation of program to CCG: The main idea in this translation are twofold: (1) build a computational graph of standard operators such as + (this step is in standard practice); (2) capture the "if-then-else" nesting within which a variable is updated (or re-updated) by introducing **if** graph nodes (there are two such nodes in Figure 3(b)). This allows us to define AD

[1]For operators beyond the basic set such as sin, log, etc., a higher value of $\mathbf{u}$ is typically used [9].

rules uniformly for both node types. To capture this nesting, we introduce the notion of a *scope* function from $Vars$ to graph nodes where $S(x)$ looks up the value of x in S. Scope updates are shown as $S[x \leftarrow E]$ where $S[x \leftarrow E](y)$ is E if $x = y$ else $S(y)$. Let S_{init} be the initial scope consisting of graph nodes for all items in Vars. We also maintain a *predicate context* $\mathbf{p}$ with rules that deal with commands, C; it is a Boolean graph node for the Boolean expression representing the path under which command C is executed. The overall effect of compiling $pgm(Vars;\ C;\ x)$ in predicate-context p and scope S_{init} is to first compile the command C starting with S_{init}, returning a final scope S_{final}, and then looking up x in it.

In our running example, variables h and g are first assigned in the initial predicate scope of predicate "True" at lines 6 and 7, generating CCG nodes $\mathbf{n4}$ and $\mathbf{n2}$. Later, line 8 containing the "if" condition creates a new predicate scope $S1$ having predicate $\mathbf{P1}$. Scope $S1$ updates the variables in the following command block. Similarly, line 10 containing the "else" creates a new predicate scope $S2$ having predicate $!\mathbf{P2}$ which updates the variables in the "else" command block. Line 13 marks the end of the inner "if-then" block which causes the predicate scope $S3$ to merge into its parent scope $S2$. This results in the creation of node $\mathbf{n9}$. Similarly, line 14 marks the end of the "if-then-else" block causing first $S1$ and $S2$ to merge together, followed by a merge into the parent scope S_{init} creating the node $\mathbf{n10}$. Finally line 15 creates the node $\mathbf{res}$ which is the output variable which we want to analyze. This procedure of translating programs to CCGs is formally defined in Appendix A.

Generating and Accumulating Error Expressions: This is the central phase in SEESAW, and consists of these sub-phases:

(a) local error analysis,
(b) instability analysis and predicate weakening, and
(c) conditional ($\bowtie$) accumulation.

We now illustrate each of these sub-phases:

(a) *Local Error Analysis:* Consider how the local error generated at node $\mathbf{n1}$, written $\mathcal{E}^{lr}(\mathbf{n1}) = (x \cdot y) \cdot \mathbf{u}$, gets propagated to the output node res. First, observe that this local error term gets propagated to the output along two paths, namely: (1) σ_1 : $(\mathbf{n1}, \mathbf{n2}, \mathbf{n5}, \mathbf{n6}, \mathbf{n10}, \mathbf{res})$, and (2) $\sigma_2 : (\mathbf{n1}, \mathbf{n2}, \mathbf{n9}, \mathbf{n10}, \mathbf{res})$. Denote the worst case impact of $\mathcal{E}^{lr}(\mathbf{n1})$ on $\mathbf{res}$, by $\mathcal{E}^{tr}(\mathbf{res}|\mathbf{n1})$. To calculate this, we need to evaluate the partial derivative $\mathbf{res}$ with respect to $\mathbf{n1}$ along each of these two paths and *add* them (this is a conservative over-approximation that ignores error cancellation which cannot be modeled during symbolic analysis). Essentially, these derivatives describe the path strength for this propagation. However, we need to obtain path-precise derivatives, and these would be modeled by a *predicated expression* that determine whether these conditional paths are "available" (turned on). In this expression, we will use the notation $[\![P \rightarrow E]\!]$ to denote the expression "if P then E."

In more detail, consider path σ_1 and apply reverse-mode differentiation starting from output $\mathbf{res}$ and flowing back to $\mathbf{n1}$ along this path as shown by Equation (3):

$$\begin{aligned}\left.\frac{\partial res}{\partial n1}\right|_{\sigma_1} &= \frac{\partial res}{\partial n10}\cdot\frac{\partial n10}{\partial n6}\cdot\frac{\partial n6}{\partial n5}\cdot\frac{\partial n5}{\partial n2}\cdot\frac{\partial n2}{\partial n1}\\ &= [\![T \rightarrow 1]\!]\cdot[\![P1 \rightarrow 1]\!]\cdot[\![T \rightarrow 1]\!]\\ &\qquad\cdot[\![T \rightarrow \frac{-1}{(x+xy)^2}]\!]\cdot[\![T \rightarrow 1]\!]\\ &= [\![P1 \rightarrow \frac{-1}{(x+xy)^2}]\!]\end{aligned} \tag{3}$$

Here, T denotes "True". Similarly, the derivative along path σ_2 is obtained as $\left.\frac{\partial res}{\partial n1}\right|_{\sigma_2} = [\![(\neg P1 \wedge \neg P2) \rightarrow 1]\!]$. Note here that we cannot accumulate the error terms along these two paths using standard conditional addition since these paths are mutually exclusive. One could argue that this simple example itself might be rescued by analyzing the impacts of the paths separately and considering the maximum error over each of the paths. However, this approach is unsatisfactory, leading to the next sub-phase of our approach now explained.

(b) *Instability Analysis and Predicate Weakening:* The *instability region* created by these two paths in our running example under floating-point semantics must be smoothly handled. The higher level idea is that when rounding error flows into two otherwise mutually exclusive predicates P and $\neg P$, the predicates develop an "overlap region." We will model this overlap by *weakening* these predicates to P^w and $(\neg P)^w$. Intuitively, weakening takes predicate expressions such as $E_1 < E_2$ and converts it to $(E_1 - err_{E_1}) < (E_2 + err_{E_2})$ where err_{E_1} is the rounding error associated with E_1 and err_{E_2} is the rounding error associated with E_2. This models the fact that rounding errors of expressions E_1 and E_2 affect a conditional. The exact manner in which weakening is defined during error analysis is involved, and hence formally described in §IV. In a sense, after weakening, we will find that $P^w \wedge (\neg P)^w$ is not false; it can be satisfiable by the overlap region of the "error-infused" predicate expression.

(c) *Conditional Accumulation:* To model accumulation over predicated paths, we introduce a new conditional accumulation operator, written $\bowtie$, formally defined as follows:

$$\begin{aligned}&[\![Pred_i \rightarrow PExpr_i]\!] \bowtie [\![Pred_j \rightarrow PExpr_j]\!] =\\ &[\![(Pred_i \wedge Pred_j) \rightarrow PExpr_i \bowtie PExpr_j\\ &\quad|\ (Pred_i \wedge \neg Pred_j) \rightarrow PExpr_i\\ &\quad|\ (\neg Pred_i \wedge Pred_j) \rightarrow PExpr_j\\ &\quad|\ (\neg Pred_i \wedge \neg Pred_j) \rightarrow 0]\!]\end{aligned} \tag{4}$$

The $\bowtie$ operator, in effect, partitions the solution domain based on the constraints involved. It takes the Boolean combinations of the predicates involved, choosing $PExpr_i \bowtie PExpr_j$ for those cases where the predicates $Pred_i$ and $Pred_j$ have an overlap. It chooses $PExpr_i$ for points in $Pred_i$ not in $Pred_j$, and $PExpr_j$ for points in $Pred_j$ not in $Pred_i$. For points outside of $Pred_i$ and $Pred_j$, 0 is chosen (modeling the absence of a path contribution).

Notice that in practice, typically $Pred_i$ and $Pred_j$ begin as mutually exclusive arms of an "if-then-else", thus modeling

the real-valued semantics. After weakening, $Pred_i$ and $Pred_j$ develop an overlap (modeling the floating-point semantics), thus making all the cases of $\bowtie$ interesting. In a sense, both the "then" and "else" develop a small overap region, thus contributing a "sneak path" to the output. In a sense, the error accumulated under the then-branch and else-branch are summed up for this overlap region. This is a conservative over-approximation of how "if-then-else" behaves under floating-point semantics.

With these changes, we can write $\mathcal{E}^{tr}(\mathbf{res}|\mathbf{n1})$ as follows:

$$\begin{aligned}
&\mathcal{E}^{tr}(\mathbf{res}|\mathbf{n1}) = \mathcal{E}^{lr}(\mathbf{n1})\left(\left.\frac{\partial res}{\partial n1}\right|_{\sigma_1} \bowtie \left.\frac{\partial res}{\partial n1}\right|_{\sigma_2}\right) \\
&= (xy)\cdot\boldsymbol{u}\cdot\left(\llbracket P1^w \rightarrow \frac{-1}{(x+xy)^2}\rrbracket \right. \\
&\qquad\qquad \left. \bowtie \llbracket(\neg P1)^w \wedge (\neg P2)^w \rightarrow 1\rrbracket\right) \\
&= \llbracket(P1^w \wedge (\neg P1)^w \wedge (\neg P2)^w) \rightarrow \\
&\qquad\qquad xy\left(1-\frac{1}{(x+xy)^2}\right)\cdot\boldsymbol{u}\rrbracket \\
&\quad \mid \llbracket(\neg P1)^w \wedge (\neg P2)^w \rightarrow -\frac{xy}{(x+xy)^2}\cdot\boldsymbol{u}\rrbracket \\
&\quad \mid \llbracket P1^w \wedge (P1^w \vee P2^w) \rightarrow (xy)\cdot\boldsymbol{u}\rrbracket
\end{aligned} \tag{5}$$

Paving Intervals, Optimization for Error and Instability: The final phase of error analysis is global optimization. Given that the final output error expression will be in terms of the input variables, we must maximize this error expression over the input domain. Finding this maximum is made difficult by the fact that in general this maximization must occur over a non-rectangular domain. To see this fact via an example, consider Figure 3(a) where the error generated at line 12 at node $\mathbf{g}$ is constrained by the values for which the conditional at line 11 (which involves non-linear comparisons) is true; and the set of these values does not span a rectangular domain.

To facilitate the use of Gelpia [9], an efficient rectangular domain optimizer, we pave the constraint domain using rectangular tiles using Realpaver. This covers the non-rectangular domain with a user-specified number of rectangular tiles "optimally" after such tiling, Gelpia can be invoked over each tile separately. This procedure helps determine the rounding error as well as instability jumps.

Using the ideas introduced thus far in this walk-through, we can obtain the following results. First, the absolute error value we get for $\mathbf{res}$ on running the example program through our tool with $x, y \in [0.01, 10.0]$ is $3.69e{-}14$. Second, the instability jump value of 7.81 for $\mathbf{res}$ can also be obtained.

IV. Formal Details of Robustness Analysis

We now describe SEESAW's algorithms rigorously, beginning with how a CCG is processed. The syntax-directed compilation of a given program to CCG is described formally in Appendix A.

We employ $\mathbf{n}$ and n to denote a functional (output value-oriented) graph node or variable, respectively, and $\mathbf{p}$ and p to denote a Boolean graph node or variable, respectively. Computational graphs $\mathbf{n}$ have the following syntax: they are a single variable node $\mathbf{n0}$ for $n0$ in $Vars$, an operator $\mathbf{op}$ applied to two graphs $\mathbf{n1}$ and $\mathbf{n2}$, or conditional nodes $\mathbf{if}(\mathbf{p}, \mathbf{n1}, \mathbf{n2})$. The Boolean expressions themselves have a Boolean computational graph $\mathbf{p}$ given by $\mathbf{p0}$ for $p0$ in $BVar$, a Boolean operator $\mathbf{bop}$ applied to $\mathbf{p1}$ and $\mathbf{p2}$, or a relational operator $\mathbf{rel}$ applied to $\mathbf{n1}$ and $\mathbf{n2}$.

```
1  function gexp(node_type)
2    switch node_type:
3      case n: return n;
4      case p: return p;
5      case !p: return ¬p;
6      case op(n1, n2):
7       return op(gexp(n1), gexp(n2))
8      case if(p, n1, n2):
9       return
10        ⟦wexpr(p) → gexp(n1)⟧ ⋈ ⟦wexpr(!p) → gexp(n2)⟧
```

Listing 1: CCG to Expression Compilation

CCG to Expression Compilation: A given CCG is interpreted as a predicated expression as described by Listing 1. We begin with a CCG node, interpret it as an operator, and recursively interpret the node's children via a recursive *gexp* call. We interpret an **if** node through a conditional sum ($\bowtie$) of the "then" and "else" expressions.

Original	Canonical
Precondition: $nonPred(E)$ E	 $\llbracket T \rightarrow E\rrbracket$
$op\ \llbracket P \rightarrow E\rrbracket$	$\llbracket P \rightarrow op(E)\rrbracket$
$\llbracket P_1 \rightarrow E_1\rrbracket\ op\ \llbracket P_2 \rightarrow E_2\rrbracket$	$\llbracket(P_1 \wedge P_2) \rightarrow (E_1\ op\ E_2)\rrbracket$
Precondition: $nonPred(E_1) \wedge nonPred(E_2)$ $\llbracket T \rightarrow E_1\rrbracket \bowtie \llbracket T \rightarrow E_2\rrbracket$	 $E_1 + E_2$
$\llbracket P_1 \rightarrow E_1\rrbracket \bowtie \llbracket P_2 \rightarrow E_2\rrbracket$	$\llbracket(P_1 \wedge P_2) \rightarrow (E_1 \bowtie E_2)$ $\mid(P_1 \wedge \neg P_2) \rightarrow E_1$ $\mid(\neg P_1 \wedge P_2) \rightarrow E_2$ $\mid(\neg P_1 \wedge \neg P_2) \rightarrow 0\rrbracket$

TABLE I: Canonicalization rules

Canonicalization rules: We canonicalize predicated expressions into a standard form so that error analysis rules can then be defined on the canonicalized expressions. The canonicalization process is captured by the rules in Table I. The first rule says that under the precondition that E is a non-predicated expression (not of the form $\llbracket P \rightarrow E\rrbracket$), then it can be treated as a predicated expression by adding a predicate T. The second rule says that a unary operator applied to a predicated expression is the same as the operator applied to E under predicate P. The third case shows how to apply a binary operator op to two predicated expressions. The fourth case models predicate accumulation when the predicates are both T (True). It says that when E_1 and E_2 themselves are non-

predicated expressions, then bowtie summation under True predicates becomes ordinary addition. The last case defines $\bowtie$ (the definition in Equation 4 is repeated for completeness).

In a sense, these canonicalization rules help us simplify predicated paths upon which reverse-mode AD is being performed. In our approach, partial derivatives can also be applied to **if** nodes, as will soon be presented in Table II. When we do that, predicated expressions are generated.

```
1  function wexpr(op(n1, n2))
2    switch op:
3      case rel: // rel ∈ {<, ≤, >, ≥}
4        return gexp(n1) − err(n1) rel gexp(n2) + err(n2);
5      case bop: // bop ∈ {∨, ∧}
6        return bop(wexpr(p1), wexpr(p2));
```

Listing 2: Weakening Predicated Expressions via function wexpr

Weakening Predicated Expressions: Weakening of predicate expressions is captured by Listing 2. It essentially modifies Boolean relational expressions formed by inequalities $<$, $\leq$, $>$ and $\geq$ by padding error terms around the inequalities so as to conservatively capture the effects of floating-point error. For this to occur, error analysis at node $\mathbf{n_1}$ and $\mathbf{n_2}$ must be done (captured by $\mathbf{err(n_1)}$ and $\mathbf{err(n_2)}$, respectively). (The error analysis underlying $\mathbf{err}(\ldots)$ is presented later in this section.) The inequalities are then suitably "corrected" (in the conservative direction) with the floating-point error. We handle $=$ and $\neq$ by converting them to suitable Boolean combinations of basic relational operators. For ordinary Boolean operations such as conjunction and disjunction, weakening is recursively applied to both operands. Negation is handled by flipping the inequality suitably (in our program syntax, all Boolean expressions BE are conjunctions and disjunctions of relational inequalities).

Graph Node	Derivative w.r.t.	Value
$+(\mathbf{n_1}, \mathbf{n_2})$	$\mathbf{n_1}$	1
$*(\mathbf{n_1}, \mathbf{n_2})$	$\mathbf{n_1}$	gexp($\mathbf{n_2}$)
$/(\mathbf{n_1}, \mathbf{n_2})$	$\mathbf{n_1}$	1/gexp($\mathbf{n_2}$)
$/(\mathbf{n_1}, \mathbf{n_2})$	$\mathbf{n_2}$	-gexp($\mathbf{n_1}$)/(gexp($\mathbf{n_2}$))^2
$\mathbf{if}(\mathbf{p}, \mathbf{n_1}, \mathbf{n_2})$	$\mathbf{n_1}$	$[\![$gexp($\mathbf{p}$)$\rightarrow 1]\!]$
$\mathbf{if}(\mathbf{p}, \mathbf{n_1}, \mathbf{n_2})$	$\mathbf{n_2}$	$[\![$gexp(!$\mathbf{p}$)$\rightarrow 1]\!]$

TABLE II: Partial Derivative Rules

Partial Derivative Rules: Building error expressions using the CCG and Equation 1 requires calculating partial derivatives. The derivative rules employed to calculate the partial derivatives of CCG graph nodes are summarized in Table II. Most of the derivative rules are familiar from calculus. The last two rules show how we treat **if** nodes as if they are operator nodes. In a sense, **if** nodes are like multiplexors. When a "conducting path" is available, the derivative strength is 1 (values are passed as such). However, the path is available only when the predicate is true. This is the reason why we model the partial derivative via a predicated expression. The "then" path retains the predicate p while the "else" path retains the negated expression $\neg p$.

```
1  function PS(no, ni)
2        if (ni == no) then return 1;
3        else
4              ps_acc := 0; // Path strength accumulator
5              for nt ∈ children(no) do
6                    ps_acc := ps_acc ⋈ (PS(nt, ni) · ∂no/∂nt);
7        return ps_acc;
```

Listing 3: Path Strength Accumulation via function PS

Path Strength Accumulation: We now define the path-strength of a node $\mathbf{n_i}$ targeting (flowing into) a node $\mathbf{n_o}$ via PS (Listing 3). Basically, "path strength" is the summation of all derivative chains. More specifically, when we perform reverse-mode AD, multiple paths can backtrack from output node $\mathbf{n_o}$ to an intermediate node $\mathbf{n_i}$.

When $\mathbf{n_o} = \mathbf{n_i}$, $PS(\mathbf{n_o}, \mathbf{n_i}) = 1$; i.e., the backtracking terminates. Otherwise, recursively backtrack from $\mathbf{n_o}$ by iterating over all children $\mathbf{n_t}$ of $\mathbf{n_o}$. We first compute the path strengths $PS(\mathbf{n_t}, \mathbf{n_i})$, and multiplying it with $\partial\ n_o\ /\ \partial\ n_t$ using the chain rule. This procedure is carried out using a *dynamic programming-style* approach as in [5].

```
1  function err(n)
2      value_acc := 0; // Error Value Accumulator
3      for ni ∈ support(n) do //support(n) are nodes whose values
           flow into n
4          value_acc := value_acc ⋈ |PS(n, ni)| · gexp(ni) · u;
5      return value_acc;
```

Listing 4: Error Value Accumulation via function err

Error Value Accumulation: Listing 4 describes error computation at any node $\mathbf{n}$ (including the final output $\mathbf{res}$): the *symbolic* output error is bounded by an expression obtained through the $\bowtie$ accumulation process. Each of the accumulated quantities represent the product of the local error at node $\mathbf{n_i}$, namely $gexp(\mathbf{n_i}) \cdot \mathbf{u}$, and $PS(\mathbf{n}, \mathbf{n_i})$. In a sense, we are multiplying local errors with path strengths and bowtie-summing (via $\bowtie$) over all available paths.

```
1  function maxInstability(res)
2      δ := 0; // Max Instability Initialized
3      for path_i ∈ paths_res do //paths_res is the set of all paths from
           leaves to the output root node.
4          for path_j ∈ paths_res AND j > i do
5              t_δ := |(gexp(res, path_i) − gexp(res, path_j))|;
6              if(t_δ > δ) δ := t_δ;
7      return δ;
```

Listing 5: Max Instability Jump Calculation

Max Instability Jump: Intuitively, the max instability is the amount by which the final output $\mathbf{res}$ jumps when we switch from one conditional path to another. This is rigorously defined by the function maxInstab($\mathbf{res}$) in Listing 5. Let $paths_{\mathbf{res}}$ be the set of all paths leading up to the root from the leaves. Let $gexp(\mathbf{res}, path_i)$ be an overloaded version of $gexp(\mathbf{res})$ specific to path i and denote the absolute value attained by the output node $\mathbf{res}$ when $path_i$ is "driven" (across all inputs for which the predicates along $path_i$ are true).

V. Evaluation

Experimental Setup: SEESAW is a Python based framework. Its symbolic data types are derived from `Symengine`. All benchmarks were executed using Python-3.8.0 on a dual 14-core Intel Xeon CPU E5-2680v4 2.60GHz CPUs (total 28 cores) with 128GB of RAM. The core analyses in SEESAW were executed without multicore parallelism (same for FPTAYLOR and PRECISA) to obtain comparable timings (the global optimizer Gelpia [10] may internally spawn multiple threads). Gelpia was used in FPTAYLOR as well as in SEESAW in an identical manner. Version v4.1406 of FLUCTUAT was used.

Benchmarks: Our benchmark suite covers a wide variety of examples with special focus on cases involving floating-point expressions impacting the control flow. Such algorithms (common in computational geometry [11] and collision detection problems [12]) are typically designed under an assumption of computations being performed in exact real arithmetic. Given the numerical and geometrical nature of these algorithms, they are particularly sensitive to numerical robustness problems. Additionally, we include benchmarks involving convergence testing such as in partial differential equation solving, Conjugate Gradient methods and Gram-Schmidt orthogonalization. We have also ported additional existing benchmarks for straight-line codes with and without branches from other tools suites such as PRECISA and FPTAYLOR.

Our evaluation of SEESAW emphasizes the following key aspects that are important when analyzing floating-point programs:

A. Worst case error bounds

We report the bounds on the maximum absolute *roundoff error* that can be observed at the program output(s). This analysis does not suppress inputs that fall into instability regions (constraints are comprised of weakened predicates).[2]

We support a basic *Optimizer* and a *Preconditioned Optimizer*. The predicates defining the function domains are set to 'True' in the basic Optimizer. This allows us to obtain a baseline where we revert back to rectangular domains encompassing the actual function domains. This is guaranteed to be a sound (if quite loose) over-approximation. To include constraints, we have experimented with two SMT solvers, namely Dreal and Z3, that can filter out boxes as needed (for reasons of space, these results are not reported). Owing to their wide spread use in Floating Point Error Analysis related projects [13], [14], our choice of SMT solvers for handling non-rectangular domains in SEESAW were Z3 and Dreal. In the Preconditioned Optimizer, we report results from a second approach called *Preconditioning* (or "paving" using RealPaver): we filter out invalid intervals a priori and feed smaller intervals that contain the valid solution space.

Table III summarizes the evaluation of benchmarks involving straight-line codes with conditionals for the two solver types. Across all benchmarks, the order of absolute error bounds reported are within one order of magnitude for each of the solver type. However, the basic Optimizer obtains near optimal answers with a short execution time in most cases, however trading off bound tightness. This trade-off is more pronounced when performing the *instability analysis* because the order of the function jumps are affected if solved without sufficiently tight domain constraints. We ran each solver with the same 'timeout' parameter of 10 seconds per query (1000s of optimizer queries made).

[2] SEESAW also supports the option to suppress such inputs; keeping these inputs gives nearly the same—and more conservative—results.

B. Empirical testing and statistical profiling

To empirically calibrate the tightness of our error estimates, we cross-checked it through shadow value calculations at higher precision over a million samples per benchmark. Also, after obtaining a predicated expression representing rounding error, we statistically sample this function over a population of intervals, thus providing an alternate view of the error unbiased by potential over-approximations that solvers can introduce.

Column 'shadow profiling' in Table III summarizes the results of the empirical testing. SEESAW's bounds were found to be always conservative and in many case within the same order of magnitude of the observed empirical bound while being on average no worse than two orders of magnitude across all solver types (except in specific cases of the basic Optimizer, which being oblivious to the domain constraints, may result in 'inf').

Instead of performing global optimization of the symbolic error expression, one can also statistically sample it. This provides a reasonable view of the underlying error symbolic expression unaffected by overapproximations introduced by optimizers. A large variation between rigorous worst-case error bound and sampled error bound during statistical evaluation can provide valuable insight to the user—especially if the worst-case is caused by a few outlier values. In particular, the bounds obtained through sampling the error expressions must lie between the rigorous worst-case bounds and the exact runtime error on concrete data points. This is because the error expressions themselves are over-approximations of true runtime rounding errors. The difference between "Prof Error Expr" and "Shadow profiling" captures the amount by which our rigorous error analysis tends to be conservative. In almost all cases, the worst-case rigorous bound is within one order of magnitude of the shadow value error bound.

In Table III, column 'Prof Error Expr' summarizes the error estimates obtained by sampling the rigorous error expression bound. Such sampling methods are often preferred to shadow-value based error estimation, which requires the same code to be instantiated in two different precisions—not always easy. Our sampling methods follow a user-specified strategy (the default being uniform sampling).

C. Instability analysis

We report the maximal potential output result-jump caused by instability. Additionally, SEESAW can sweep across all potential instability sites and rank order the amount of result jump that can potentially be introduced at those sites.

Benchmarks	Worst case absolute error estimation				Instability Jump		Empirical Profiling		
	Optimizer		Preconditioned Optimizer						
	Error	ExecTime	Error	ExecTime	Optimizer	Preconditioned Optimizer	Prof Error Expr	Shadow profiling	Instability jump
Barycentric[a]	1.04e-15	39.00s	1.04e-15	38.00s	7.81	7.81	8.24e-16	4.6e-16	0.93
SymSchur[a]	4.13e-16	5.50s	4.02e-16	44.00s	0.42	0.02	8.56e-17	2.3e-17	1.48e-08
DistSinusoid[a]	**inf**	–	7.17e-06	23.40s	**inf**	3.17	2.56e-16	2.18e-15	0.05
Interpolator[c]	7.49e-14	1.34s	1.47e-14	0.58s	225.00	33.25	1.14e-14	5.32e-15	11.25
Nonlin-Interpol	2.24e-14	0.40s	4.44e-16	1.12s	101.00	2.93	2.77e-16	3.9e-16	2.9
EnclosingSp[a]	4.41e-15	1.67s	4.41e-15	7.31s	12.63	4.19	1.89e-15	5.10e-16	4.74e-07
cubicspline[c]	8.88e-15	1.60s	4.16e-16	1.32s	27.00	0.25	1.94e-16	2.7e-17	0.25
linearfit	3.07e-16	1.02s	3.07e-16	1.04s	1.08	0.318	7.5e-17	2.58e-16	0.13
jetApprox[c]	7.45e-15	3.45s	7.04e-15	3.16s	15.13	5.79	1.90e-15	1.64e-15	0.20
SqDistPtSeg[a]	6.81e-13	7.50s	5.60e-13	42.16s	317.58	150.63	5.9e-14	7.13e-14	7.14
Squareroot[b]	1.77e-15	1.28s	1.77e-15	0.18s	2.68	2.68	6.10e-16	3.06e-17	–
Styblinski[c]	4.09e-14	2.01s	2.02e-14	1.31s	51.62	0.01	7.25e-15	3.30e-16	3.21e-07
Test2[a]	7.90e-14	2.34s	7.65e-14	15.00s	101.90	3.02	2.43-14	2.07e-14	0.69
ClosestPoint[a]	9.98e-16	6.30s	9.15e-16	24.00s	4.59	1.45	5.56e-16	5.15e-16	2.6e-07
SpherePt[a]	6.43e-15	66.00s	6.16e-e15	255.00s	0.87	0.82	1.71e-15	1.87e-16	1.02e-06
lead-lag[b]	2.49e-16	6.98s	2.49e-16	12.50s	0.04	0.14	8.23e-17	3,21e-18	1.51e-04
EigenSphere[a]	1.67e-15	480.00s	1.67e-15	540.00s	2.16	2.16	6.09e-16	4.4e-16	1.37e-07
Jacobi[a]	4.82e-13	22.00s	4.09e-13	151.00s	81.33	55.50	5.69e-14	2.96e-14	–
Mol-Dyn[a]	2.94e-15	138.00s	2.94e-15	137.00s	2.11	2.11	1.39e-15	1.22e-15	5.67e-06
Gram-Schmidt[b]	2.42e-16	99.00s	2.42e-16	226.00s	0.02	0.02	3.97e-17	3.88e-17	8.37e-04
Ray Tracing[a]	4.70e-14	331.00s	4.65e-14	317.00s	87.70	0.129	1.42e-14	1.59e-16	2.90e-08
ExtremePoint[a]	1.69e-14	30.00s	1.69e-14	33.00s	115.50	115.50	1.68e-14	1.0e-19	4.52e-06
SmartRoot[b]	**inf**	–	2.44e-15	0.89s	**inf**	0.52	3.90e-16	0	1.26e-07
Poisson[a]	1.0e-17	120.00s	1.0e-18	121.00s	0.97	0.97	3.7e-18	2.33e-19	–

[a] New benchmarks introduced with SEESAW for conditional codes
[b] Benchmarks ported from FPBench
[c] Benchmarks ported from PRECiSA

TABLE III: SEESAW evaluation for worst case error bounds and instability reporting.

While SEESAW's CCG generation facilitates obtaining guarded path constraints identifying valid program paths, the path conditions may end up containing many atomic inequalities containing large floating-point expressions. We rigorously analyze each of these Boolean predicates involved in path conditions and weaken each individual constraints by including the instability region ("gray zone") introduced due to roundoff errors using the $wexpr$ rule in Listing 2. Suppose P represents the original atomic inequality and P' denotes the weakened constraint obtained by 'weakening' P to include the rounding error. Under this transformation, $(P' \wedge \neg P')$ does not evaluate to 'False'. Instead, it identifies exactly the instability region in which the real execution and the floating-point executions may diverge and cause instability.

For a Boolean predicate composed of many such inequalities, obtaining the cover for the combined instability region is in general non-trivial. However, applying our *predicate weakening* approach to every atomic formula lends itself easily to being incorporated in the most general manner as part of the domain constraints. We solve for the function difference along each program path pairs in these instability regions and report the maximum for *instability quantification*. The instability expression must be solved only for the instability constraint, otherwise it would be reporting bloated results over the entire interval as shown in Table III. In fact, we observe that the instability values become orders of magnitude tighter even with RealPaver guided PreConditioned intervals.

We also performed *empirical testing* to explore the input domain over randomly sampled points to hit regions that trigger these instabilities. In Table III, the last column lists the maximum instability jump observed empirically using 10M tests per benchmark. The '-' entries did not hit any instability issues in our empirical tests.

a) **Instability Bound Ranking***:* SEESAW also reports the program path pairs that resulted in control flow divergence, and hence helping root-cause the source of maximum instability. Although SEESAW reports the instability bound for the output node, it additionally tracks the localized instability information at every internal node of the conditional graph. It can yield additional analytical information on instability by rank-ordering these bounds for every node and extracting out the corresponding path predicates. For example, consider the 'Barycentric benchmark' which computes triples of numbers that defines the position of a point in reference to a triangle containing the point. In [12], the code for Barycentric coordinates requires evaluating two predicates. We list the outer predicate $P1$ and the inner predicate as $P2$. Each of these predicates makes decisions on the dot-product evaluation for pairs of sides of the reference triangle – making them susceptible to floating-point inaccuracies. SEESAW estimates an instability bound at the output of 7.81 if the control-flow traverses the instability region. This analysis also exposes the predicates causing instability of internal nodes, giving a chance for the user to rewrite these predicates in their code. Table IV shows the rank-ordering of predicate combinations based on the maximum instability variations seen across every internal

and output node.

Impacted Node	Predicates	Instability
Internal	$[P2', \neg P2']$	215.2665
Internal	$[P2' \wedge \neg P1' \wedge (P2' \vee \neg P2')\ ,$ $P1' \wedge \neg P2' \wedge (P2' \vee \neg P2')]$	215.2665
Output	$[P1', P2' \wedge \neg P1']$	7.81
Internal	$[\neg P1' \wedge \neg P2', P2' \wedge \neg P1']$	2.74
Internal	$[P1', \neg P1' \wedge \neg P2' \wedge (P2' \vee \neg P2')]$	0.058

TABLE IV: Instability Bound Ranking: All impacted nodes are listed with the associated predicates and the instability values. In particular, the output node **res** is driven by 2 paths δ_1 and δ_2.

D. Comparative analysis

In Table V we present comparative results for SEESAW, PRECISA and FLUCTUAT for benchmarks containing straight-line codes with conditions. Comparison is presented for the total execution time, absolute error bounds and the bounds obtained on the output instability. We report statistics obtained when using the 'Preconditioned Optimizer' solver type for SEESAW.

We observe that while PRECISA obtains better runtimes in few of the benchmarks in comparison to SEESAW, it is more efficient only for smaller examples (it times out on many of SEESAW's larger examples). In contrast, SEESAW obtains tight error bounds for all benchmarks except in one (Barycentric)[3].

FLUCTUAT reports the best execution times across this set of benchmarks. This is one of the best aspects of FLUCTUAT which is designed to handle larger pieces of codes. SEESAW's focus is on scaling adequately while offering rigorous and tighter bounds.

In examples such as Symmetric Schur (SymSchur) decomposition, SEESAW produces bounds that are 2 orders of magnitude tighter than obtained by FLUCTUAT, and 10 orders of magnitude tighter than produced by PRECISA. SEESAW is able to handle larg and complex benchmarks with thousand of operators and multiple conditionals nests.

We ran a memory profiler on the benchmarks when running both PRECISA and SEESAW. Memory profiling of SEESAW and PRECISA over Barycentric, Gram-Schmidt and Spherepoint revealed that SEESAW had a consistent memory usage, averaging 80MB over the entire execution. PRECISA's memory requirements went up from 150 MB for Barycentric to 4GB (spiking to over 8GB) for Gram-Schmidt, and 1GB to 5GB for Spherepoint, terminating in both cases.

We also performed a tool comparison amongst SEESAW, PRECISA, FPTAYLOR and FLUCTUAT to ascertain performance consistency over straight-line programs without conditionals. PRECISA obtains better execution times in about 50% of the cases in which it completes, particularly those with few dozen operators. However, for large straight-line codes ($>$100 operators), both PRECISA and FPTAYLOR run into scalability issues. SEESAW performs consistently well, in execution speed and bound tightness across all these benchmarks and produces the tightest bounds. FLUCTUAT has the best overall execution speed. However, it bottlenecks for larger and complex programs (the FFT benchmark in our case), quitting unexpectedly. As said earlier, FLUCTUAT is a closed-source tool whose innards are unknown to us. FLUCTUAT only reports the presence of instability jumps—not its magnitude.

[3]These bounds get tightened when used with constrained solver options – using Z3/Dreal with increased query limit

VI. DISCUSSIONS, RELATED WORK, CONCLUSIONS

While many automatic differentiation approaches exist for error analysis [15], [16] that achieve significant scalability, our symbolic automatic differentiation is based on *symbolic* analysis, and thus able to handle intervals of input values ([15], [16] work over specific values). Being able to provide analysis results over intervals of inputs is basic to providing rigorous guarantees.

Many other rigorous analysis tools for floating-point error analysis have recently been proposed including Gappa [17], Real2Float [18], Rosa [13], FPTAYLOR [9], Numerors [19], and even specialized tools for cyberphysical systems [20], [21]. The importance of offering *formally certified* bounds using theorem proving is also well-recognized [9], [22].

Coming to programs that contain loops, FLUCTUAT does include support for conditionals *and* loops. However, this comes with the obligation of users having to define abstract domains to assist the tool. Darulova [1] gives methods to handle loops; however this work has not handled programs containing loop-bodies as involved as those in our benchmarks.

Handling loops while generating tight and meaningful error bounds is a significant item of future work that remains in front of the community. This may require non-trivial loop invariant estimation methods as well as fixpoint computing methods based on *policy iterations* [23].

SEESAW efficiently handles examples involving a variety of complex calculations such as computing EigenSpheres. Internally, such examples require computing covariance matrices and the Symmetric Schur decomposition. SEESAW reports much tighter bounds for both floating-point errors and instability jumps with an efficient memory usage profile. It offers practical levels of scalability as is clear from our benchmarking. The work in [20], [21] offers another rigorous approach supported by theorem-proving.

Many tools (e.g., FPTAYLOR [9], SATIRE [5], PRECISA [4]) in this area use a global optimizer to estimate the upper bound of roundoff error. These optimizers work over rectangular input domains using variants of branch-and-bound [9], [10], [24].

For loop-free programs with conditionals, input domains are, in general, non-rectangular, and their shape is dictated by the nature of the conditional expressions. Work based on Zonotopes (e.g., FLUCTUAT) tends to lose precision in case the conditionals contain non-linear predicates (which they often do in most of our benchmarks). SEESAW's symbolic reverse-mode AD smoothly handles this situation through its idea of

Benchmarks	OPs	Conds	Absolute Error			Exec Time[n]		Instability		
			SEESAW	PRECISA	FLUCTUAT	SEESAW	PRECISA	SEESAW	PRECISA	FLUCTUAT
squareroot[b]	5	1	1.77e-15	4.20e-08	**1.11e-16**	0.18s	**0.01s**	**2.68**	2.70	Y
Barycentric[a]	83	2	**1.04e-15**	1.05e-14	2.67e-15	**39.00s**	30.00m	**3.93**	4.26	Y
SymSchur[a]	23	2	**4.02e-16**	1.30e-04	2.21e-14	**44.00s**	1750.00s	**0.02**	0.44	Y
DistSinusoid[a]	32	2	**7.17e-06**	NA	1.27	**23.40s**	–	**3.17**	–	Y
Interpolator[c]	22	4	**1.47e-14**	3.07e-14	1.51e-14	1.34s	**0.10s**	**33.25**	225.00	Y
Nonlin-interpol[c]	6	1	**4.44e-16**	3.19e-14	3.19e-14	1.12s	**0.08s**	**2.9**	101.01	Y
EnclosingSphere[a]	35	3	**4.41e-15**	1.67e-07	9.77e-15	**7.00s**	9.80s	**4.19**	9.18	Y
Test2[a]	19	3	**7.65e-14**	3.53e-08	6.55e-12	18.00s	**0.60s**	**3.02**	51.80	Y
Closestpt2[a]	31	2	**9.15e-16**	4.24e-14	4.06e-15	**24.00s**	31.00s	**1.45**	17.35	Y
Cubicspline[c]	44	4	**2.66e-15**	1.33e-14	1.99e-15	1.60s	**0.40s**	**0.25**	27.00	Y
Styblinski[c]	55	3	**7.32e-15**	4.04e-14	2.91e-14	**1.90s**	43.00s	**0.01**	93.40	Y
SpherePt[a]	75	3	**1.76e-15**	–	1.78e-14	**40.00s**	–	**0.36**	–	Y
lead-lag[b]	46	8	**2.93e-17**	–	2.81e-16	**11.00s**	–	**0.01**	–	Y
RayTracing[a]	119	8	**1.46e-15**	–	inf	**331.00s**	–	**0.12**	–	Y
EigenSphere[a]	334	13	**6.36e-16**	–	2.07	**438.00s**	–	**2.16**	–	Y
Jacobi[a]	170	2	**1.15e-13**	–	8.10e-12	**138.00s**	–	**55.50**	–	Y
Mol-Dyn[a]	179	6	**1.42e-15**	–	6.21e-15	**–**	–	–	–	Y
Gram-Schmidt[b]	471	19	**7.95e-17**	–	1.44e-16	**115.00s**	–	**0.02**	–	Y
Poisson[a]	2736	4	**1.00e-19**	–	–	**20.00s**	–	**0.97**	–	–
CG[a]	210K	4	**1.86e-18**	–	–	**3142.00s**	–	–	–	–

[a] New benchmarks introduced with SEESAW for conditional codes
[b] Benchmarks ported from FPBench
[c] Benchmarks ported from PRECISA

TABLE V: Comparative analysis with PRECISA for codes with conditional branches. [n]: FLUCTUAT took < 0.1 sec for all the rows. '-' denotes program timed out.

predicate weakening, and our results corroborate this tightness advantage.

Our work is the first we know that employs the idea of paving to deal with non-rectangular domains and still be able to use a backend rectangular domain optimizer. In terms of solver technologies, Rosa [13] propagates errors in numeric affine form and uses SMT solvers to obtain tight bounds.

The idea of weakening predicates is mentioned in previous work (e.g., Darulova [1]). However, in our work, we provide the first implementation (we know) of these ideas within a symbolic reverse-mode AD framework.

VII. Concluding Remarks

Scalable and rigorous analysis of floating-point code is of increasing importance in a number of application domains. In this paper, we plug one major hole in this area: the lack of a scalable open-source tool for this task applicable to codes coming from practical domains. We provide a rigorous definition of reverse-mode automatic differentiation with conditionals naturally incorporated. We provide thorough empirical testing of our results and also a statistical profiling method for error expressions.

In conclusion, SEESAW is a rigorous robustness analysis tool that can play a central role during the design of critical software systems that include conditionals. These could for instance be geometrical libraries that employ different algorithms to compute the same quantity for different input ranges. A designer can also use SEESAW to analyze the many codes already in the field where one uses ad hoc methods such as "self-adjusting paddings." An example one often finds is an expression such as $x < y$ being into another expression such as $x < (y + f(x, y))$. The rationale is to make the separation between x and y get adjusted by a "fudge factor" captured by the expression $f(x, y)$. Today, there aren't any tools one can readily use to assess the gains of (or problems due to) such fudge-factors. Tools such as SEESAW can help designers arrive at such heuristic code adjustments backed with the reassurance of robustness analysis. In fact, designers will be able to *publish rigorous specifications* including rounding errors and instability jumps for their codes, thanks to the use of tools such as SEESAW. This practice can go a long way toward supporting code reuse and portability [25].

References

[1] "Programming with Numerical Uncertainities," 2014. [Online]. Available: https://people.mpi-sws.org/~eva/papers/thesis.pdf

[2] K. Ghorbal, E. Goubault, and S. Putot, "A Logical Product Approach to Zonotope Intersection," in *Proceedings of the 22nd International Conference on Computer Aided Verification*, ser. CAV'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 212–226. [Online]. Available: https://doi.org/10.1007/978-3-642-14295-6_22

[3] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine, "Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software," in *Formal Methods for Industrial Critical Systems, FMICS 2009*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, vol. 5825, pp. 53–69.

[4] L. Titolo, M. A. Feliú, M. Moscato, and C. A. Muñoz, "An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs," in *Lecture Notes in Computer Science*. Springer International Publishing, Dec. 2017, pp. 516–537. [Online]. Available: https://doi.org/10.1007/978-3-319-73721-8_24

[5] A. Das, I. Briggs, G. Gopalakrishnan, S. Krishnamoorthy, and P. Panchekha, "Scalable yet Rigorous Floating-Point Error Analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[6] "SymEngine," https://github.com/symengine/symengine/, 2019.

[7] L. Granvilliers and F. Benhamou, "Algorithm 852: RealPaver: An Interval Solver Using Constraint Satisfaction Techniques," *ACM Trans. Math. Softw.*, vol. 32, no. 1, p. 138–156, Mar. 2006. [Online]. Available: https://doi.org/10.1145/1132973.1132980

[8] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, Mar. 1991. [Online]. Available: http://doi.acm.org/10.1145/103162.103163

[9] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 1, Dec. 2018. [Online]. Available: https://doi.org/10.1145/3230733

[10] "Gelpia: A Global Optimizer for Real Functions," 2017. [Online]. Available: https://github.com/soarlab/gelpia

[11] M. Berg, O. Cheong, M. Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008.

[12] C. Ericson, *Real-Time Collision Detection*. USA: CRC Press, Inc., 2004.

[13] E. Darulova and V. Kuncak, "Sound Compilation of Reals," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2014, pp. 235–248.

[14] S. Gao, S. Kong, and E. M. Clarke, "dReal: An SMT Solver for Nonlinear Theories over the Reals," in *Proceedings of the 24th International Conference on Automated Deduction, CADE 2013*, 2013, pp. 208–214.

[15] H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger, "ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[16] M. O. Lam, T. Vanderbruggen, H. Menon, and M. Schordan, "Tool integration for source-level mixed precision," in *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, 2019, pp. 27–35.

[17] M. Daumas and G. Melquiond, "Certification of Bounds on Expressions Involving Rounded Operators," *ACM Transactions on Mathematical Software*, vol. 37, no. 1, pp. 1–20, 2010.

[18] V. Magron, G. Constantinides, and A. Donaldson, "Certified Roundoff Error Bounds Using Semidefinite Programming," *ACM Transactions on Mathematical Software*, vol. 43, no. 4, pp. 34:1–34:31, Jan. 2017. [Online]. Available: http://doi.acm.org/10.1145/3015465

[19] M. Jacquemin, S. Putot, and F. Védrine, "A Reduced Product of Absolute and Relative Error Bounds for Floating-Point Analysis," in *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, ser. Lecture Notes in Computer Science, A. Podelski, Ed., vol. 11002. Springer, 2018, pp. 223–242. [Online]. Available: https://doi.org/10.1007/978-3-319-99725-4_15

[20] M. M. Moscato, L. Titolo, M. A. Feliú, and C. A. Muñoz, "Provably correct floating-point implementation of a point-in-polygon algorithm," in *Formal Methods – The Next 30 Years*, M. H. ter Beek, A. McIver, and J. N. Oliveira, Eds. Cham: Springer International Publishing, 2019, pp. 21–37.

[21] R. Salvia, L. Titolo, M. A. Feliú, M. M. Moscato, C. A. Muñoz, and Z. Rakamarić, "A Mixed Real and Floating-Point Solver," in *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, ser. Lecture Notes in Computer Science, J. M. Badger and K. Y. Rozier, Eds., vol. 11460. Springer, 2019, pp. 363–370. [Online]. Available: https://doi.org/10.1007/978-3-030-20652-9_25

[22] H. Becker, N. Zyuzin, R. Monat, E. Darulova, M. Myreen, and A. Fox, "A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4," in *FMCAD*, 10 2018, pp. 1–10.

[23] A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot, "A policy iteration algorithm for computing fixed points in static analysis of programs," in *Computer Aided Verification*, K. Etessami and S. K. Rajamani, Eds. Springer Berlin Heidelberg, 2005.

[24] J.-M. Alliot, N. Durand, D. Gianazza, and J.-B. Gotteland, "Finding and Proving the Optimum: Cooperative Stochastic and Deterministic Search," in *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI)*. ACM, 2012, pp. 55–60.

[25] D. H. Ahn, A. H. Baker, M. Bentley, I. Briggs, G. Gopalakrishnan, D. M. Hammerling, I. Laguna, G. L. Lee, D. J. Milroy, and M. Vertenstein, "Keeping Science on Keel When Software Moves," *Commun. ACM*, vol. 64, no. 2, p. 66–74, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3382037

APPENDIX
FORMAL GENERATION OF CCG FROM SOURCE CODE

The rules in Figure 5 underlying SEESAW's program syntax guide the translation of program to CCG. *Exp1*, *Exp2* and *Exp3* define the generation of operator nodes ($\mathbf{n1}$) and input nodes ($\mathbf{x}$) in the CCG (likewise *BExp1* and *BExp2*). Rules *Cmd1*, *Cmd2*, *Cmd3*, *Cmd4* and *Cmd5* are command rules that are fired in response to the program structure such as assignment operation, "if-then" or "if-then-else" structures and define the updates to the *scope*. The *Merge* rule defines the scope update when the program switches from a child scope to the parent scope. The command C is the program to be compiled, and is handled by rule $Exp1$, generating the root of the CCG $\mathbf{n}$. This command is analyzed by rules $Cmd1$ through $Cmd5$, appealing to the expression rules Exp and $BExp$ as is the case. Rule $Merge$ helps create **if** nodes.

$$(Exp1)\ \frac{(\mathbf{p}, C, S_{init}) \Longrightarrow^* (\mathbf{p}, skip, S_{final}) \quad S_{final}(out) = \mathbf{n}}{(pgm(Vars; C; out), S_{init}) \Longrightarrow^* \mathbf{n}}$$

$$(Exp2)\ \frac{true}{(x, S) \Longrightarrow \mathbf{S(x)}}$$

$$(Exp3)\ \frac{(E1, S) \Longrightarrow \mathbf{n1},\ (E2, S) \Longrightarrow \mathbf{n2}}{(E1\ op\ E2, S) \Longrightarrow \mathbf{op(n1, n2)}}$$

$$(BExp1)\ \frac{(E1, S) \Longrightarrow \mathbf{n1} \quad (E2, S) \Longrightarrow \mathbf{n2} \quad \mathbf{p = rel(n1, n2)}}{(E1\ rel\ E2, S) \Longrightarrow \mathbf{p}}$$

$$(BExp2)\ \frac{(BE1, S) \Longrightarrow \mathbf{p1} \quad (BE2, S) \Longrightarrow \mathbf{p2} \quad \mathbf{p = bop(p1, p2)}}{(BE1\ bop\ BE2, S) \Longrightarrow \mathbf{p}}$$

$$(Cmd1)\ \frac{(E, S) \Longrightarrow \mathbf{n}}{(\mathbf{p}, x := E, S) \longrightarrow (\mathbf{p}, skip, S[x \leftarrow \mathbf{n}])}$$

$$(Cmd2)\ \frac{(\mathbf{p}, C1, S) \Longrightarrow (\mathbf{p}, C1', S')}{(\mathbf{p}, C1; C2, S) \longrightarrow (\mathbf{p}, C1'; C2, S')}$$

$$(Cmd3)\ \frac{true}{(\mathbf{p}, skip; C2, S) \longrightarrow (\mathbf{p}, C2, S)}$$

$$(Cmd4)\ \frac{(BE, S_{main}) \Longrightarrow \mathbf{p1},\ \mathbf{p2 = and(p1, p)} \quad (\mathbf{p2}, C, S_{main}) \longrightarrow^* (\mathbf{p2}, skip, S')}{(\mathbf{p}, if\ BE\ then\ C, S_{main}) \longrightarrow^* (\mathbf{p}, skip, merge(\mathbf{p1}, S', S_{main}))}$$

$$(Cmd5)\ \frac{\begin{array}{c}(BE, S_{main}) \Longrightarrow \mathbf{p1} \quad \mathbf{p2 = and(p1, p)},\ \mathbf{p3 = and(!p1, p)} \\ (\mathbf{p2}, C1, S_{main}) \longrightarrow^* (\mathbf{p2}, skip, S') \quad (\mathbf{p3}, C2, S_{main}) \longrightarrow^* (\mathbf{p3}, skip, S'')\end{array}}{(\mathbf{p}, if\ BE\ then\ C1\ else\ C2, S_{main}) \longrightarrow^* (\mathbf{p}, skip, merge(\mathbf{p1}, S', S''))}$$

$$(Merge)\ \frac{S_1(x) = \mathbf{n_1} \text{ and } S_2(x) = \mathbf{n_2}}{\mathbf{if(p, n_1, n_2)} \in merge(\mathbf{p}, S_1, S_2, x)}$$

Fig. 5: Translation of Programs to CCGs