FPyDebug: Numerical Accuracy Profiling

Brett Saiki
bsaiki@cs.washington.edu
University of Washington
Seattle, Washington, USA

Wei Jun Tan
wj428@cs.washington.edu
University of Washington
Seattle, Washington, USA

Abstract

Floating-point rounding errors easily lead to unacceptable deviations from the intended real number computation, yet they remain difficult to debug. Current numerical accuracy debugging tools suffer from two distinct problems: they are external from the language ecosystem—users must cycle between the implementation environment and the debugging environment—and they rely on approximate methods—errors may not be caught or valid code is falsely reported. Both of these issues discourage user of these tools and decrease their reliability. Instead, we propose such debugging should be lightweight—available within the environment itself—and accurate—using exact oracles when possible.

We implemented our ideas in FPyDebug, a numerical accuracy profiler that can detect if a function suffers from numerical error, and if so, ranks the function's expressions by the likelihood they contribute to the observable error. To measure accuracy correctly, FPyDebug provides real number computation, extending prior work to work with statements and complex control flow. To be convenient, FPyDebug is available as a library in Python for Python code. We demonstrate FPyDebug on 134 benchmarks from across a wide variety of domains. FPyDebug allows users to easily and accurately identity numerical errors within their program. Moreover, its real number evaluation strategy is more accurate than approximate methods, incurring a 2.9× performance overhead.

1 Introduction

Detecting and debugging floating-point rounding errors remains a difficult task. These errors may be subtle or silent, yet cause computations to unacceptably deviate from the intended real number behavior. Worse, programming environments are increasingly heterogeneous: new accelerators offer a large selection of number formats, each with different precision and memory bandwidth trade-offs, and new instructions offering numerical capabilities beyond simple arithmetic, often with unusual rounding behaviors. Developers must leverage these features and navigate complex trade-offs to maximize the performance of their programs without compromising numerical accuracy, while inevitably encountering bugs including numerical errors.

The current landscape of numerical accuracy debugging tools is insufficient. These tools are often *external*: users must load their binary into a debugging tool, diagnose problems within the tool's environment, and manually interpret the output to repair their code. For example, prior work [2, 22] use the Valgrind framework [16] while others [4, 5] utilize compiler instrumentation like AddressSanitizer [23]. Additionally, they rely on *approximate* methods. Their oracles use arbitrary-precision floating-point [2, 22] to increase the precision of intermediate computations, or error-free transformations [5], to preserve accuracy with additional error terms; neither technique guarantees the result is sufficiently close to the ideal computation. Thus, these tools suffer from long debugging loops, discouraging programmers from using them, or inaccurate results, producing time-consuming false positives, or worse, not detecting problems at all. Without easily usable numerical accuracy debuggers, programmers may often perturb code haphazardly or resort to unnecessarily large precision to fix or simply mask poor numerical behavior, incurring both accuracy and performance costs.

To address these issues, we introduce FPyDebug, a *numerical accuracy profiler* that is *lightweight* and *accurate*. FPyDebug provides two modes: (i) a *function accuracy profiler* that detects if a function suffers from numerical errors, and (ii) an *expression accuracy profiler* that ranks expressions based on the error they introduce even when their free variables have no accumulated error, i.e., a local measure of numerical error. These two profilers take the perspectives of the caller and callee, respectively: how a function may affect accuracy when its result is used in future computations, and what accuracy issues within a function may impact the accuracy of the return values. Both of these profilers leverage a *real number evaluator* based on the Rival interval library [11], to compute the correctly-rounded real number value to measure the actual deviation of the floating-point computation at any point in a program. Moreover, we implement FPyDebug as a Python library to debug Python code directly; thus, providing a convenient debugger that may interact with the broader Python ecosystem.

We demonstrate FPyDebug on a set of 134 numerical benchmarks from the FPBench suite [7]. We show that FPyDebug allows a user to easily and accurately identify numerical errors within their programs. Moreover, FPyDebug's profilers are more accurate than approximate methods like high-precision floating-point computation seen in prior work. We then illustrate that real number evaluation is only 2.9× slower than approximate methods and produces a numerical result 90% of the time.

This paper contributes the following:

- Two accuracy profilers to evaluate which functions have numerical error and which expressions likely contribute to the overall error (Section 3).
- A real number evaluator that computes the correctly-rounded real number result of programs with statements and complex control flow (Section 4).

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- An implementation in Python to analyze numerical programs in an embedded DSL (Section 5).

Section 2 provides necessary background on floating-point numbers, rounding error, and arbitrary-precision interval arithmetic. Section 6 evaluates FPyDebug on programs with numerical errors. Section 7 discusses FPyDebug's limitations and suggests future work to address them. Section 8 covers related work, and Section 9 concludes.

2 Background

We provide necessary background for understanding FPyDebug. First, we describe floating-point numbers (Section 2.1) and numerical error (Section 2.2). Then, we cover arbitrary-precision interval arithmetic, a technique for real number computation (Section 2.3).

2.1 Floating-Point Numbers

Floating-point numbers ($\mathbb{F}$) are the subset of real numbers ($\mathbb{R}$) representable in scientific notation:

$$(-1)^s \times c \times 2^{exp},$$

where $s \in \{0, 1\}$, c is an integer with precision p, i.e., c is representable in p digits, and exp is any integer. Most literature on floating-point numbers represents these numbers in"normalized" form,

$$(-1)^s \times 1.m \times 2^e,$$

where m is a digit sequence of length $p-1$, and $e = exp-p+1$, so that c scales appropriately to $1.m \in [1, 2)$. [1] The IEEE 754 standard [1] describes a common representation of floating-point number that may be encoded in a fixed number of bits. To do so, each IEEE 754 floating-point format has a range of valid exponent values $[exp_{\min}, exp_{\max}]$ and a maximum precision $p_{\max}$ that restricts the set of representable values in the format. The IEEE 754 standard reserves a small subset of non-real values in each format, namely $\pm\infty$, and not-a-number error values.

Each floating-point operation must produce a floating-point result based on the ideal real number result. Formally, each floating-point operation $\hat{f} : \mathbb{R}^n \to \mathbb{F}$ defines $rnd : \mathbb{R} \to \mathbb{F}$, the *rounding mode* of $\hat{f}$, a function that maps each ideal real number result x to a floating-point number $rnd(x)$. Then, the floating-point operation is exactly:

$$\hat{f}(x) = rnd(f(x))$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the ideal mathematical operation.

The most common rounding mode is "round to nearest, ties to even" (RTE) which uses the following rules:

- if $x \in \mathbb{F}$, then $rnd(x) = x$,
- if $x \notin \mathbb{F}$ and $\hat{x} \in \mathbb{F}$ is uniquely the closest floating-point value to x, then $rnd(x) = \hat{x}$;
- otherwise, there exist $x_1, x_2 \in \mathbb{F}$ that are equidistant to x, and $rnd(x)$ is whichever value has an even mantissa m.

For fixed-size formats, if $\hat{x}$ is larger in magnitude than the largest representable in the format $+\hat{x}_{\max}$, then $\hat{f}$ produces either $\pm\infty$ or $\pm\hat{x}_{\max}$ (with the sign of $\hat{x}$), depending on the rounding mode. This condition is called *overflow*. The opposite condition, when $\hat{x}$ is too small in magnitude is called *underflow*; however, it requires no special treatment in our presentation of floating-point numbers.

Reversing the rounding operation, we obtain

$$rnd^{-1}(\hat{x}) = \{x \in \mathbb{R} \mid rnd(x) = \hat{x}\},$$

the interval of real values that will round to $\hat{x}$. We say that $rnd^{-1}(\hat{x})$ is the *rounding envelope* of $\hat{x}$. Under the RTE rounding mode, we compute the rounding envelope of a floating-point number $\hat{x}$ in the following way. Identify $x_1, x_2 \in \mathbb{F}$, the nearest floating-point numbers below and above $\hat{x}$ ($x_1 < \hat{x} < x_2$). Then $rnd^{-1}(x) = [x_l, x_h]$ where $x_l = (x_1+\hat{x})/2$ and $x_h = (x_2+\hat{x})/2$ are the midpoints between x_1 (or x_2) and $\hat{x}$.

2.2 Rounding Error

Rounding error is the numerical difference between the floating-point result and the ideal real number result of a function (or program). For an individual operation, the *relative error* ε comes from the rounding operation: $\hat{x} = rnd(x) = x + x\varepsilon$. Rounding error introduced by an individual operation come in two flavors: insufficient precision or overflow (underflow). If x requires more than p digits (or possibly an infinite number of digits), $rnd(x)$ introduces an error of up to 2^{-p} times smaller than $\hat{x}$. If x is too large, then $\hat{x}$ is either $\pm x_{\max}$ or $\pm\infty$ which introduces potentially an unbounded amount of numerical error. Similarly, if x is too small, $\hat{x} = 0$ and the relative error is 1.

In practice, only basic arithmetic guarantees a relative error of 2^{-p}, also called the *machine epsilon*, assuming no overflow or underflow. Transcendental operations like exponentiation and trigonometric functions from common math libraries (like `math.h` in the C language), may introduce far more error. Typically, it is a small integer multiple more than the machine epsilon.

Rounding errors also accumulate as the intermediate floating-point values drift from the ideal real number value. In addition to introducing rounding error, operations may amplify errors in their arguments. Consider $(x + 1) - x$. Ideally, this expression would always compute 1. However, for large floating-point values, say $x \approx 10^{16}$, $x + 1$ rounds to x, so $(x + 1) - x = 0$. While the initial rounding error of $x + 1$ is relatively small (machine epsilon), the accumulated error causes a complete loss in accuracy, often called catastrophic cancellation.

2.3 Arbitrary-Precision Interval Arithmetic

When computing an accurate estimate of the real number result is crucial, standard floating-point computation is often insufficient as it introduces potentially unbounded relative error. Prior work [2, 4, 22] rely on *arbitrary-precision floating-point* computation, where the precision p may be increased to tens of thousands of digits, to approximate the real number result with higher confidence. While this technique may overcome problems with rounding error in many cases, any computed value may still suffer from unbounded relative error and its distance to the ideal real number value is unknown.

Rather than using floating-point numbers, interval arithmetic represents the result of each computation as an interval $[lo, hi]$ and ensures that the result of every floating-point operation $\hat{f}$ contains the true real number result. The invariant of *sound* intervals is

[1] Floating-point standards like the IEEE 754 standard also support subnormal numbers where the significand $1.m$ is instead represented by $0.m$.

crucial: the composition of multiple operations still produces an interval containing the ideal result. In practice, errors accumulate and the interval becomes impractically wide. However, by combining interval arithmetic with arbitrary-precision floating-point computation, we can recompute at increasingly higher precision until the interval becomes sufficiently small or the intermediate precision becomes too large and the computation fails. Prior work [11] explores how to quickly recognize when increasing precision will not cause the interval to converge.

For the purposes of this paper, we will define sufficiently small to be the condition when there exists a (unique) floating-point number x such that $rnd(lo) = x$ and $rnd(hi) = x$ under the RNE rounding mode. We say that x is the correctly-rounded result; or that x is accurate to p digits of precision. This condition is equivalent to the interval being contained within the rounding envelope of x, i.e., $[lo, hi] \subseteq rnd^{-1}(x)$.

3 Accuracy Profilers

FPyDebug provides two accuracy profilers: a *function profiler* which determines if a function's floating-point result differs from the real number result; and an *expression profiler* which identifies which expressions may be causing such errors. Both profilers compare how floating-point execution numerically deviates from the ideal computation, just at different granularity: entire function or individual expressions, respectively. These profilers rely on a real number evaluator (Section 4), which computes the closest floating-point number to the real number result at a specified precision.

3.1 Function Accuracy Profiler

The function profiler identifies when functions have *observable* numerical error, when its output deviates from the real number result. The distinction of observability is important since rounding error can accumulate *within* the function but have little effect on the return value. For example, the naive expression

$$x^2 + (x + 1) - x$$

has the familiar subexpression $(x + 1) - x$ from Section 2.2. The subexpression should ideally be 1, but the floating-point result becomes 0 for large x. However, when $(x + 1) - x$ becomes 0, x^2 is much larger than 1, so adding 1 has no effect since it is rounded off. Therefore, the function has minimal observable rounding error. If the profiler detects large numerical error for many inputs, than we should interpret the result at any call site of the function to be $y + \varepsilon$ where y is the ideal result and ε is a noticeably large error. In this case, the developer may wish to probe further with the expression profiler (Section 3.2), invoke a repair tool like Herbie [18, 20, 21], or rewrite their code themselves.

Figure 1 shows the algorithm for the function profiler. Given a function f of floating-point (and boolean) operations, the function profiler evaluates $\hat{y_i} = \hat{f}(x_i)$ on a set of input points $x_1, ..., x_n$ using the usual IEEE 754 floating-point semantics and compares the result to the real number value of $y_i = f(x_i)$, rounded to p_{max} digits, the maximum number of digits allowed by the type of $\hat{y_i}$. This precision requirement is important since it ensures that the real evaluator produces the closest value to the real number result that is the *same* type as the floating-point result, i.e., a implementation with no observable error with the same type signature would produce this value. The values y_i and $\hat{y_i}$ may differ due to accumulated numerical error. The profiler aggregates these errors for a set of input points, producing either the average of errors or the maximum error.

```
def function_profile(f, xs):
  errs = []
  float_eval = NativeEvaluator()
  real_eval = RealEvaluator()
  for x in xs:
    y_float = float_eval.run(f, x)
    y_real = real_eval.run(f, x, prec(y_float))
    err = error(y_fl, y_real)
    errs.append(errs)
  return summarize(errs)
```

Figure 1: Function profiler. Runs a function `f` on a set of input points `xs`, computing the difference between the floating-point result `y_float` and real-number result for each `y_real` input value. Reports the aggregate error, either by averaging or selecting the worst-case error.

FPyDebug allows the user to specify which error function used to measure the error between the floating-point and real number result. As long as the error function correctly ranks errors by their severity, FPyDebug's results are reasonable. We present a few common error metrics. The *absolute error* $|y - \hat{y}|$ simply measures the positive difference between the two values while the relative error $|(y-\hat{y})/y|$ (discussed in Section 2.1) measures the positive difference scaled such that the magnitude of y has no effect. Herbie [18, 20, 21] uses *ordinal error* which approximately measures the number of floating-point values between y and $\hat{y}$, transformed by $\log_2(x + 1)$. The intuition for this metric is it estimates the number of "better" floating-point values the function could have returned with $ord_{\text{err}}(y, \hat{y}) = 0$ iff $y = \hat{y}$.

3.2 Expression Accuracy Profiler

While the function accuracy profiler measures the observable numerical error of a function for a set of input points; the expression accuracy profiler identifies *internal* numerical error. This finer-grained profiling can identify when the floating-point computation of an expression deviates from the real number result, even when the value of each free variable is *correctly rounded*. This technique adapts the *local error heuristic* introduced in Herbie [18] which measures the floating-point error introduced by *individual operations*. The local error heuristic is finer-grained than expression profiling, but likely suffers from scaling issues; FPyDebug operates on programs rather than small expressions, so expression profiling is better suited.

Figure 2 shows the algorithm for the function profiler. Like the function profiler, the algorithm uses both the floating-point and real number evaluator. For each input point `x` the real evaluator runs the function on `x`, recording for each expression `e`, the computed value `y_real` and the environment, the mapping from variables to values, captured immediately before executing the expression. After the real evaluator executes, we evaluate each recorded expression `e`

```
def expr_profile(f, xs):
  errs_by_expr = {}
  float_eval = NativeEvaluator()
  real_eval = RealEvaluator()
  for x in xs:
    for expr, y_real, env in real_eval.run_with_trace(f, x) :
      y_float = float_eval.run(expr, env)
      err = error(y_float, y_real)
      errs_by_expr[expr].append(err)
  return { expr: summarize(errs) for expr, errs in errs_by_expr }
```

Figure 2: Expression profiler. Runs a function `f` on a set of input points `xs`, identifying expressions where the floating-point and real-number result deviate. Reports the aggregate error, either by averaging or selecting the worst-case error.

using the floating-point evaluator and the environment of *correctly-rounded* values to produce `y_float`. The profiler measures the error between the real number and floating-point result and records it. Finally, the errors are aggregated for expression as with the function profiling, computing the average error or finding the maximum error for each expression.

Using the environment from the real evaluator at each program point rather than from the floating-point evaluator is an important choice. It ensures that the values passed to the floating-point evaluator have minimal error; they are correctly rounded. Thus any numerical error measured after executing the expression must be *solely* from the expression itself. This observation agrees with the principle underlying the local error heuristic introduced by Panchekha et al. [18], which considers error coming solely from an operation. It should be noted, however, that since the intermediate values come from the real number evaluator, the computed floating-point values may differ those that would be computed during normal execution. This observation identifies another key difference between the function and expression profiler: if the expression profiler identifies an expression of interest, it may be the case that the expression neither introduces any real error during normal floating-point evaluation (since the values might be already perturbed) *nor* contributes to any observable rounding error outside of the function (recall the example in Section 3.1). Rather, the measured error is an estimate of how likely the expression is contributing to the overall observable rounding error.

4 Real Number Evaluator

Both accuracy profilers, the function profiler (Section 3.1) and the expression profiler (Section 3.2), require comparing the result of floating-point computations against a real number oracle. One such technique for real number oracles is *arbitrary-precision floating-point arithmetic* as implemented in FPDebug [2], PositDebug [4], and Herbgrind[22] (see Section 2.3). Another technique is *arbitary-precision interval arithmetic* which actually attempts to compute the real number computation, failing only when the required internal precision exceeds a set limit. FPyDebug uses arbitrary-precision interval arithmetic for its accuracy, at the expense of possible failures, since it guarantees that its result is the closest floating-point value to the real result.

An oracle using arbitrary-precision floating point computes with floating-point arithmetic using some sufficiently large precision $k \gg p$, where p is the desired precision of the output, in hopes that the high precision result is accurate up to p' digits where $p' \geq p$. However, arbitrary-precision floating-point arithmetic still suffers from numerical error, so the condition $p' \geq p$ is not guaranteed. For even simple expressions, the required precision may be large; $\sqrt{x+1} - \sqrt{x}$ at $x = 10^{300}$ requires over 1000 binary digits of precision to ensure the result is accurate to at least the same precision as double-precision floating-point values. On the other hand, an oracle using *arbitrary-precision interval arithmetic* like in Rival [11] produces an interval containing the real result; the condition $p' \geq p$ is equivalent to the endpoints of the interval rounding to the same value at precision p (see Section 2.3). FPyDebug extends Rival, which only evaluates *expressions*, to work on larger programs with statements and complex control flow.

FPyDebug's real number evaluator computes on functions with boolean and floating-point operations with assignment statements, conditional statements, while loops, and return statements. For a function f with formal arguments $x_1, \ldots, x_n$, the real evaluator requires input floating-point values $v_1, \ldots, v_n$ for each argument, and an output precision p_{out}. The evaluator produces a result that is the ideal real number up to p_{out} digits. Importantly, all intermediate values are either *intervals* containing the real number result, a guarantee from Rival's sound interval invariant, or a boolean value. We will denote by $v = \texttt{rival}(e, p)$ an evaluation by Rival of an expression e such that v is correctly rounded to at least p digits of precision. This call does not guarantee a result; if the expression requires excessively high precision for an intermediate operation, then Rival fails and an exception is thrown.

To understand how FPyDebug extends Rival, we will first consider the case where the only statements are assignment statements `x_i = e_i` and return statements `return e_r`. Each e_i represents some expression of floating-point or boolean operations, literals, and variables To simplify our analysis, the return statement must appear as the last statement of the function. The key challenge is for any expression e_i and precision p_i, the evaluation $v_i = \texttt{rival}(e, p)$ may fail because the interval of some previous computation may be too wide; and thus, the current computation requires more precision, beyond the limit, to make up for the uncertainty. A naive solution is to simply use a uniformly high precision $p \gg p_{\text{out}}$ throughout the computation; however, this leads to slow performance especially when high precision is unnecessary.

We can reframe this challenge as a *precision allocation* problem: finding the minimum required intermediate precision p_i of x_i such that the final expression e_r may be computed to p_{out} digits of precision. This problem is difficult: the required precision p_i at each statement is dependent on both the precision of previously computed local variables x_j ($j < i$) and the accuracy of any future computation.

FPyDebug's real number evaluator uses a simple precision assignment algorithm. First, it allocates exactly p_{out} digits of precision to the expressions at each assignment statement. In the best case, this

```
def real_eval(f, x, p):
  # initialize precision allocation
  prec_map = {}
  for expr in f.exprs():
    if expr.has_type_real():
      prec_map[expr] = p
  # evaluation loop
  while TRUE:
    try:
      return internal_eval(f, x, prec_map)
    except PrecisionError as e:
      for expr in f.exprs():
        if expr.is_before(e.expr):
          prec_map[expr] *= 2
          if prec_map[expr] > MAX_PRECISION:
              raise PrecisionLimitExceeded()
```

Figure 3: Real evaluation. Evaluate a function `f` on input `x` at precision `p`. After allocating precision, the evaluator iteratively tries computing with the current precision assignments (`internal_eval`). When Rival fails, precision assignments for previous expressions are doubled. Boolean expressions are not assigned precisions.

is sufficient and each evaluation $\texttt{rival}(e_i, p_i)$ succeeds. In many cases, some intermediate computation, say at e_j, fails due to excessive intermediate precision. Using the hypothesis that the failure was due to a local variable x_k $(k < j)$ having too little precision, the evaluator doubles the allocated precision at each statement preceding `x_j = e_j` and restarts computation from the first statement. This loop continues until either the evaluator produces a result without encountering a precision exception, or the evaluator allocates precision beyond the limit set by Rival. In this case, the evaluator has no recourse; it too raises a precision exception that the user must handle.

Extending this technique to conditional statements and loops is mostly straightforward. Conditional statements and loops both have a single expression evaluating to a boolean value to indicate which statement to execute next. Since the return type is a boolean, precision need not be assigned—this is only relevant to real numbers. For both of these syntactic constructs, the precision for statements within each branch (or loop body) are reallocated based on program order. For example, if Rival fails for a statement within an if-true branch of an if statement, the precision is increased for previous statements in the branch, but precision in the if-false branch is untouched. When a failure occurs beyond a join point in the program, e.g., the point after an if statement, precision must be increased along both branches. Figure 3 summarizes our description of FPyDebug's real number evaluator.

5 Implementation

FPyDebug is a dynamic analysis tool for numerical programs in Python, providing both accuracy profiling and real number evaluation. We implemented FPyDebug as a plugin within the FPy framework [2] (Section 5.1). FPyDebug comprises of 1178 lines of Python code. FPyDebug requires the Rival library for its real number evaluator; its implementation ensures sound interchange of intervals (Section 5.2) and optimizations to avoid unnecessary loss of precision (Section 5.3).

To use FPyDebug, a programmer writes a numerical program in Python and adds the `@fpy` decorator to load it into the FPy environment. Then, possibly in the same module, the programmer constructs either of FPyDebug's profilers, `FunctionProfiler` or `ExpressionProfiler`, and runs the profiler on a list of input points. Each profiler returns a summary, either the function's observable error or a ranking of expressions by how much error they might introduce.

5.1 The FPy Language

FPy is an embedded domain specific language (DSL) within Python intended for specifying and simulating numerical programs for a wide variety of number formats. Programs in FPy are a subset of Python. The only valid datatypes are booleans, real values, and tuples. Expressions include the boolean operations, arithmetic operations, the standard set of trascendental operations seen in most math libraries, and basic tuple operations like construction, indexing, etc. FPy programs have assignment statements, conditional statements, while loops, for loops, return statements, and assert statements.

```
@fpy
def whetsone1(n: int):
    t = 0.499975
    x1, x2, x3, x4 = (1.0, −1.0, −1.0, −1.0)
    for _ in range(n):
        x1 = (x1 + x2 + x3 − x4) * t
        x2 = (x1 + x2 − x3 − x4) * t
        x3 = (x1 − x2 + x3 + x4) * t
        x4 = (−x1 + x2 + x3 + x4) * t
    return x1, x2, x3, x4
```

A Python function satisfying these restrictions may be transformed into an FPy program by simply adding an `@fpy` decorator. The example above is the first of eleven Whetstone benchmarks [6] translated to Python and marked as an FPy program. Executing this function, for example with `whetstone1(10)`, runs this function in the FPy runtime (software floating-point numbers) rather than Python's native runtime (hardware floating-point numbers). The FPy runtime allows the programmer to change the rounding behavior of the function, for example, using single-precision floating-point values [3] instead of the double-precision floating-point values to see how less precision affects overall accuracy. FPy defines a single intermediate representation (IR) for its programs, so a programmer may impelement their own interpreters, transformation passes, and program analyses. We implemented FPyDebug's real evaluator as an interpreter in FPy and FPyDebug's profilers as program analyses for FPy programs.

[2] https://github.com/bksaiki/fpy

[3] Python does not support single-precision floating-point natively.

5.2 Soundness

Importantly, intermediate numbers within FPyDebug's real number evaluator are intervals that *always* contain the actual result. FPyDebug must ensure this invariant when invoking functions from an external interval library like Rival. By default, Rival is insufficient as an interface for handling *interval* values. Although it internally maintains interval representations during computation, Rival ultimately rounds the result to the specified precision p, discarding the underlying interval structure. This rounding is *unsound*: the computed value is the nearest floating-point value to the real result which deviates from the real result by some unknown amount.

To address this issue, we modified Rival to accept real intervals as input and produce intervals as output. Therefore, neither Rival nor FPyDebug prematurely converts an interval to a floating-point value, avoiding the issue of unsound approximation. As a result, Rival's sound interval guarantee may be extended to FPyDebug.

5.3 Optimizations

To avoid unnecessary loss of precision via evaluation using Rival, FPyDebug incorporates a number of basic optimizations.

Constants. Constants are preserved exactly when possible rather than being evaluated. Some constants are not representable in floating-point, so evaluation via Rival would convert the constant into an interval with non-zero uncertainty. By deferring evaluation until the constant is actually used in a computation, FPyDebug ensures that its precise value is preserved as long as possible, minimizing unnecessary interval widening.

Copy/Constant-Propagation. In addition, FPyDebug applies copy propagation to avoid using Rival when possible. When an assignment $x = y$ occurs, the value of y is copied directly to x without invoking Rival. Avoiding unnecessary rounding helps maintain accuracy while improving efficiency. For similar reasons, FPyDebug applies this technique for assignment of constants, e.g., $x = c$.

Exactness. Another optimization is tracking exactness of intermediate variables. When Rival determines that an expression evaluates to an exact result, it produces a point interval, rather than the rounding envelope for that value. During the precision assignment algorithm, FPyDebug's real number evaluator does not need to recompute the expression at higher precision since increasing precision only shrinks intervals: exact intervals are thus fixed.

6 Evaluation

We evaluate three research questions.

(1) Does FPyDebug allow users to easily and correctly identify numerical error in programs? (Section 6.1)
(2) Can FPyDebug find sources of numerical error more accurately than approximate methods like arbitrary precision floating-point arithmetic? (Section 6.2)
(3) How often does FPyDebug's real number evaluator successfully compute the real number result? (Section 6.3)

We performed all experiments on Ubuntu 24.04.1 with an Intel Core Ultra 7 155H CPU and 16GB of RAM. We used Python 3.13 to run FPyDebug and used Racket 8.14 to build Rival.

Our evaluation uses all 134 benchmarks from the FPBench repository. [4] The benchmarks are drawn from a variety of sources, including textbooks on scientific computing and numerical analysis, prior research papers on error detection, examples of code from control devices or physics, and more. The benchmarks exhibit varying characteristics, ranging from small mathematical expressions, to large programs (30 lines) with complex control flow like loops.

6.1 Case Study: Quadratic Formula

We analyze a representative case study drawn from the 134 benchmarks from FPBench. It demonstrates how a user would use FPyDebug to diagnose numerical error in their programs. Most importantly, FPyDebug allows the user to (i) remain entirely within Python, allowing for tighter debugging loops and interfacing with the broader Python ecosystem; and (ii) measures numerical error accurately compared to approximate methods.

The case study is the usual quadratic formula (positive root only) which computes a zero of a quadratic polynomial. We restrict the function to inputs representing a polynomial with at least one root, so it returns a valid, numerical answer. We can specify this precondition as an argument to the `@fpy` decorator.

```
1  @fpy(pre=lambda a, b, c: b * b >= 4 * a * c)
2  def quadratic_pos(a, b, c):
3      d = sqrt(b * b - 4 * a * c)
4      return (-b + d) / (2 * a)
```

Diagnosis. First, we investigate if this implementation is accurate using FPyDebug's function accuracy profiler. For convenience, FPyDebug provides sampling methods for FPy function: the generated inputs satisfy the precondition and are uniformly distributed along the set of valid floating-point values rather than real numbers. The function profiler (see Section 3.1) produces a report with the average error, maximum error, and number of inputs that FPyDebug's real evaluator failed on. For this case study, we use the *ordinal error* metric mentioned in Section 3.1. Figure 4 shows the actual Python code using FPyDebug to profile the initial implementation. The profiler reports that the function has an average ordinal error of 28 (out of 64), roughly translating to having only 6 (of 17) correct decimal digits on average.

```
1  @fpy(pre=lambda a, b, c: b * b >= 4 * a * c)
2  def quadratic_pos(a, b, c):
3      d = sqrt(b * b - 4 * a * c)
4      return (-b + d) / (2 * a)
5
6  profiler = FunctionProfiler()
7  inputs = sample_function(quadratic_pos, 256, seed=1)
8  report = profiler.profile(quadratic_pos, inputs)
```

Figure 4: Quadratic Formula. A naive implementation under test with FPyDebug using 256 sampled input points. The function suffers from multiple sources of numerical error.

[4] https://github.com/FPBench/FPBench/tree/494219a6289b471e24d02ce40de3f8db06f9d884

Switching to FPyDebug's expression profiler (see Section 3.2), we see that both expressions (line 3 and 4) introduce numerical error (replace `FunctionProfiler` with `ExpressionProfiler`). The profiler reports the introduced ordinal error averages 22 and 14, respectively. Thus, our implementation of the quadratic formula suffers from at least two separate causes of numerical error.

Interoperability. Recall that we implemented FPyDebug as a Python library for analyzing Python code. With little effort, we can integrate FPyDebug with other Python libraries. Using plotting libraries like `matplotlib`, the output of FPyDebug's expression profiler may be visually displayed for each expression with a histogram.

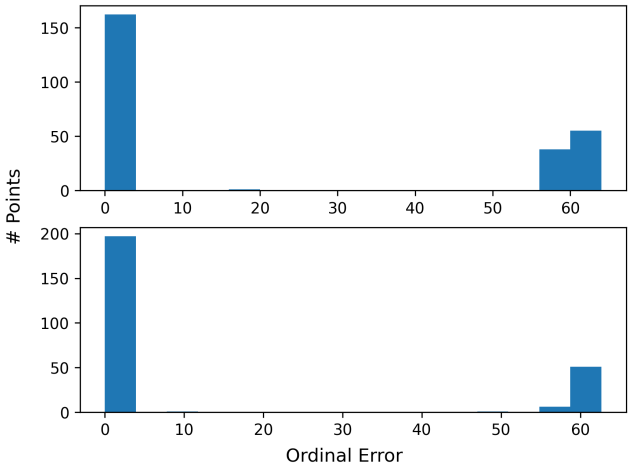

Figure 5: Combining FPyDebug's expression profiler with `matplotlib`. For each expression, the profiler measures the ordinal error introduced for either `sqrt(b * b - 4 * a * c)` (top) or `(-b + d) / (2 * a)` (bottom). Across all function inputs, there is either minimal or significant accuracy loss.

Figure 5 shows the results with the histogram for the expression `sqrt(b * b - 4 * a * c)` on top and the histogram for the expression `(-b + d) / (2 * a)` on bottom. For both expressions, there is either minimal or significant accuracy loss, with few cases between the two extremes. Thus, while the function may compute the right value about half the time, the numerical error it suffers from is often catastrophic. Importantly, visually identifying this trend required a minimal amount of work; it required only ten additional lines of code. Compared to prior work, FPyDebug is intended to be lightweight: users can more easily integrate numerical error detection from FPyDebug with other libraries for their own debugging purposes.

Repair. FPyDebug does not provide any method of *repairing* numerical error in programs, but FPyDebug can check that any repaired programs no longer suffer from numerical error. A user may invoke an automated numerical repair tool such as Herbie [18], or rely on numerical analysis expertise to fix the problem on their own. For example, Herbie produces, for our initial implementation, the following program:

```
1  @fpy(pre=lambda a, b, c: b * b >= 4 * a * c, spec=quadratic_pos)
2  def quadratic_pos_herbie(a, b, c):
3    if b <= -4e+123:
4      t = -b / a
5    elif b <= 3.05e-84:
6      t = (-b + sqrt(((b * b) - ((4 * a) * c)))) / (2 * a)
7    else:
8      t = -c / b
9    return t
```

Meanwhile, Panchekha [17] describes a method of fixing the quadratic equation that is superior to Herbie's solution.

```
1  @fpy(pre=lambda a, b, c: b * b >= 4 * a * c, spec=quadratic_pos)
2  def quadratic_pos_panchekha(a, b, c):
3    x = sqrt(abs(a)) * sqrt(abs(c))
4    if copysign(a, c) == a:
5      d = sqrt(abs(b/2) - x) * sqrt(abs(b/2) + x)
6    else:
7      d = hypot(b/2, x)
8    if b < 0:
9      t = (d - b/2) / a
10   else:
11     t = -c / (b/2 + d)
12   return t
```

We can use FPyDebug to give useful feedback even for experts in numerical analysis. Note that the `spec` keyword in the `@fpy` decorator, when provided, overrides the FPy function FPyDebug uses to compute the expected real result. This feature is especially important when an implementation introduces approximations like series expansions which may actually change the real number behavior of the function. In this case, the `spec` keyword ensures we compare against the real number semantics of the original program, e.g., `quadratic_pos`, rather than (possibly different) real number semantics of a program under test, e.g., `quadratic_pos_herbie` or `quadratic_pos_panchekha`. FPyDebug's function profiler reports that Herbie's implementation has an average ordinal error of 9.8 (compared to 28 of the original implementation); it reports that Panchekha's implementation has an average ordinal error of just 0.2.

6.2 Arbitrary-Precision Floating Point

Prior work [4, 7, 22] relies on a technique called arbitrary-precision floating-point computation for detecting numerical error; this technique simply approximates the real number result by using floating-point at sufficiently high precision. FPyDebug uses real number evaluation instead to compute the true real number result up to a given number of digits. While slower than arbitrary-precision floating point, real number evaluation does not suffer from number errors that may cause arbitrary-precision floating point to not detect numerical error or incorrectly identify numerical error when it too suffers from accuracy problems.

We analyze how well these techniques identify numerical errors on 134 FPBench benchmarks. For each benchmark, we sampled 30 points uniformly on the set of input points considered valid according to the benchmark's precondition. This sampling method, uniform sampling of the representation, differs from uniformly sampling the (subset) of the real number line (hypercube); It is based

on the observation that floating-point values are not uniformly distributed over real numbers; and thus, better covers the set of floating-point values that may actually be seen.

For each benchmark and its set of input points, we evaluated FPyDebug's expression profiler on each input point with four different real number references: FPyDebug's real number evaluator and an arbitrary-precision floating-point evaluator at 1024, 2048, and 4096 digits of precision uniformly. The arbitrary-precision floating-point is provided by the FPy runtime. We aggregated per-expression error estimates across the input points for both techniques.

We judge an expression to have numerical error when its mean ordinal error is greater than 3 (an ordinal error of $log_2(10) \approx 3.3$ corresponds to the last decimal digit being incorrect). For each real number reference, we classify each expression according to the following categories:

- *True Positive (TP) / Negative (TN).* The profiler correctly identifies that the expression has / does not have error.
- *False Positive (TP) / Negative (TN).* The profiler incorrectly identifies that the expression has / does not have error.
- *Unknown Positive (UP) / Negative (UN).* The profiler identifies that the expression has / does not have error, but we cannot establish if it is correct or not.

Since the real number evaluator produces the ideal real number, it serves as the oracle of whether an expression suffers from numerical error or not. When the real number evaluator fails, we cannot check if errors reported by the profiler (for any real number reference) are actually errors for that benchmark. For these benchmarks, we classify an error reported by the profiler as unknown positive (UP), or unknown negative (UN) if no error is reported.

Technique	TP	FP	FN	TN	UP	UN
Real	42	0	0	236	0	0
$p = 1024$	40	0	2	235	13	124
$p = 2048$	41	0	1	235	13	124
$p = 4096$	41	0	1	236	13	124

Figure 6: Comparing expression profiling with different real number references: FPyDebug's real number evaluator arbitrary-precision floating-point at 1024, 2048, and 4096 digits of precision. Arbitrary-precision reports a false negative twice at 1024 digits but only once for 2048 and 4096 digits. Real number evaluation fails for 6 benchmarks which produces a number of unknown positives or negatives.

Figure 6 shows the results by expression for each of the four real number references over the 134 benchmarks from FPBench. Overall, the results are similar: the arbitrary-precision floating-point references report only two, one, and one false negative, respectively. Thus, FPyDebug's real number evaluator is slightly more accurate than floating point, an approximate method. However, FPyDebug's real number evaluator fails for six benchmarks. As a result, we cannot classify 137 expressions profiled by the floating-point interpreters. We note that we skipped 13 additional benchmarks: 3 had infinite loops (intended for static analysis tools), 3 had FPy tensor operations (FPyDebug does not support tensor operations), 4 timed out after 5 minutes for the real number evaluator, and 3 timed out for the arbitrary-precision floating-point evaluators.

The accuracy gain of real number evaluation over arbitrary-precision floating point is marginal for these benchmarks. While simple examples may be constructed to show how these real number references deviate, these benchmarks do not suffer much from these cases. However, we note that the mismatch in the total number of expressions is due to the real number reference *choosing* a different branch than the floating-point interpreter. In addition, the profiler using floating-point produced a false positive when comparing for individual input points, but too few to affect the aggregate view. These observations together suggest a user would prefer approximate methods like arbitrary-precision floating point for numerical accuracy profiling unless accuracy is absolutely required. In these cases, arbitrary-precision floating point would be insufficient since it may choose the wrong branch, incorrectly identity numerical error, or not identity numerical error at all.

6.3 Real Number Evaluation Performance

We analyze how FPyDebug's real number evaluator behaves on 134 FPBench benchmarks, specifically its running time and how often it successfully evaluates a function for an input point. Recall from Section 4, that FPyDebug's real number evaluator may raise an exception to signal that an intermediate computation required excessive precision to compute accurately, beyond a limit imposed by the Rival library. The running time measures the overhead imposed over arbitrary-precision floating-point, while the percentage of successful evaluations measures reliability.

For each benchmark, we sampled 256 points uniformly on the set of input points considered valid according to the benchmark's precondition using the same sampling methods as described in Section 6.2. For each benchmark and its set of sampled input points, we ran FPyDebug's real number evaluator and classified the evaluator as successful if it produced a real number result or failed if it raised a precision exception. For each successful evaluation, we measured the performance overhead of real number evaluation over arbitrary-precision floating-point uniformly at 1024, 2048, 4096 digits of precision. We separate the benchmark suite into two groups: benchmarks without loops and benchmarks with loops.

Benchmark	Total	Skipped	Success Rate	Overhead
loopless	114	0	96.5%	2.57, 2.91, 2.64
loops	21	10	24.8%	13.1, 15.4, 14.5
all	134	10	90.2%	2.56, 2.87, 2.63

Figure 7: Running FPyDebug's real number evaluator on 134 benchmarks organized by benchmark type. We list the number of benchmarks and number of benchmarks skipped for each group (no points evaluated successfully). We also list the percentage of sampled points evaluated successfully and the performance overhead over arbitrary-precision floating-point at 1024, 2048, and 4096 digits, respectively.

Figure 7 shows the results of running FPyDebug's real evaluator on the benchmarks. FPyDebug's real number evaluator is highly successful on benchmarks without loops (21), producing a

real number result, rather than precision exception, for 96.5% of sampled points. In addition, the performance overhead for this evaluation method over arbitrary-precision floating-point evaluation is 2.57, 2.91, 2.64 times, respectively. However, for benchmarks with loops (21), the results are worse: FPyDebug's real number evaluator only successfully produces a real number result 24.8% of the time and imposes a 13.1, 15.4, 14.5 times overhead when it is successful. Out of the 21 benchmarks, 10 were skipped: 3 had infinite loops (intended for static analysis tools), 3 had FPy tensor operations (FPyDebug does not support tensor operations), 4 did not evaluate over the sampled points under a 5 minute timeout.

The FPyDebug's real number evaluator works well for smaller programs with a small number of statements (1 to 10 statements). For these programs, real number evaluation has a relatively low failure rate and performance overhead, so it serves a reasonable real number oracle for profiling numerical accuracy. However, the real number evaluator scales poorly in terms of the number of expressions it must evaluate with Rival. Poor scaling explains the higher failure rate and larger performance overhead for benchmarks with loops: the necessary precision increases when the number of total operators increase. We believe that this trend would be visible in large straight-line programs. For these programs, a user may prefer approximate methods like arbitrary-precision floating-point for numerical accuracy profiling.

7 Limitations and Future Work

FPyDebug works on FPy functions, rather than general Python functions, so it is limited to the set of features that FPy supports: common statements like conditional statements, while and for loops; operators for only boolean and real (integer or float) values. Improving FPyDebug's applicability requires expanding the subset of features of Python that FPy supports. In addition, the success rate of FPyDebug's real evaluator is based on the tightness of intervals returned by the underlying interval library. Research in improving interval libraries would greatly enhance the FPyDebug's usefulness in practical applications.

Neither FPyDebug nor Rival is guaranteed to be sound; any defect may cause FPyDebug's real number oracle to produce an incorrect result and misidentify rounding error. Our evaluation relied on sampling methods to produce representative inputs to functions. Improved sampling methods, including different sampling distributions, may classify numerical errors differently. Finally, while we ran FPyDebug on a large set of benchmarks, it would be useful to see how FPyDebug would be used in broader applications like scientific computing or machine learning. A user study or field study is needed to determine how beneficial FPyDebug is for actual programmers.

8 Related Work

This paper is about dynamically detecting numerical error in Python programs using a real-number oracle based on arbitrary-precision interval arithmetic to detect numerical errors.

Dynamically Detecting Numerical Error

The most common technique in dynamically detecting numerical error is to instead use additional floating-point precision as a proxy for the true real number value. FPDebug [2], Herbgrind [22], and PositDebug [4] all perform shadow execution using high-precision floating-point computation; the first two tools leverage the Valgrind framework while the last one uses compile-time instrumentation. Another approach to increasing precision are error-free transformations (EFTs) which extends each floating-point number with additional error terms. Shaman [10] relies on EFTs to dynamically track rounding error, recording the number of significant digits for each computation. EFTSanitizer [5] combines EFTs with compile-time instrumentation techniques from PositDebug. Other dynamic approaches include proxy metrics that do not directly measure actual floating-point error in a program. For example, Herbie [18, 20, 21] relies on local error heuristic to estimate an operator's individual contribution of numerical error; it localize repair efforts on regions of a program with high local error. HiFPTuner [13] combines dependence analysis and edge profiling with community detection algorithms to identify operators and variables where precision tuning may be applied to improve numerical accuracy.

Statically Bounding Error

In contrast to dynamic analyses for detecting numerical error, static analysis techniques attempt to bound the expected numerical error of a program via symbolic techniques such that the error bound is sound, yet maximally narrow. These techniques include classical numerical analysis methods, optimization, formal verification methods, and type systems. In numerical analysis, FPTaylor [24] leverages symbolic Taylor expansions which propagate symbolic error terms through a program to compute tight error bounds of floating-point expressions. Later, Satire [9] extends the techniques of FPTaylor by adding path strength reduction, rewriting error bound expressions, and replacing subexpressions by "summary" nodes. These additions allow Satire to scale to expressions that are many orders of magnitude larger than the expressions that FPTaylor can capably handle. Traditional optimization techniques may be applied to statically detecting numerical error. For example, Real2Float [15] applies semi-definite programming to compute error bounds on polynomial expressions. Other approaches to statically detecting numerical error include formal verification methods like SMT solving and theorem proving. Rosa [8] compiles real number functions with preconditions on inputs and accuracy postconditions to finite-precision floating-point code; it relies on an SMT solving to prove that the produced code soundly meets the required accuracy bound. Precisa [25] analyzes floating-point programs to produce symbolic error bounds along with formal proof certificates. More recently, Numerical Fuzz [14] is a type system for bounding numerical error of functions: function types track the maximal relative error they introduce and how error in the inputs are amplified. Notably, when a function type checks, the type system guarantees the declared output error bound is always met.

Real Number Evaluation

While real number computation is undecidable in general, there is significant prior work in doing so approximately to a sufficient degree or exact in specific cases. MPFR [12] is the state-of-the-art arbitrary-precision floating-point library that is available in many programming languages; it guarantees correct rounding of

many arithmetic and transcendental functions at a user-specified precision. MPFI [19] introduces arbitrary-precision interval arithmetic which uses intervals with arbitrary-precision floating-point endpoints. Each of its operations soundly round outwards: if the arguments to an interval function contain the intended real number value, the output interval must always contain the true real number result. Rival [11] introduces movability flags to arbitrary precision interval arithmetic to detect when increasing the working precision will not tighten the resulting interval to a given target width. FPyDebug also uses arbitrary-precision interval arithmetic, extending techniques from Rival [11] to evaluate programs with statements instead of just expressions. Finally, Boehm proposes an API for real numbers by combining rational arithmetic with recursive real arithmetic; these ideas are now implemented in Google's Android calculator.

9 Conclusion

FPyDebug introduces two accuracy profilers for identifying numerical error in floating-point programs: a function profiler to measure deviation in the floating-point and real number results of functions, and an expression profiler to find expressions that are likely contributing to such observable errors. To do so, FPyDebug leverages a real number evaluator to compute the correctly rounded result which can be used as an accurate oracle. We demonstrated FPyDebug helps a user identify numerical errors easily in a case study. We then showed that FPyDebug more accurately finds numerical errors than approximate methods like arbitrary precision floating-point computation. Finally, we showed that FPyDebug's real evaluator successful evaluates 90% of the time with an average overhead of 2.9× over arbitrary-precision floating-point computation.

Acknowledgments

We thank Michael Ernst and James Yoo for their insightful feedback on earlier versions of this paper.

References

[1] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. doi:10.1109/IEEESTD.2019.8766229

[2] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A Dynamic Program Analysis to Find Floating-point Accuracy Problems *(PLDI '12)*. ACM, New York, NY, USA, 453–462. doi:10.1145/2254064.2254118

[3] Hans-J. Boehm. 2020. Towards an API for the real numbers. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 562–576. doi:10.1145/3385412.3386037

[4] Sangeeta Chowdhary, Jay P. Lim, and Santosh Nagarakatte. 2020. Debugging and Detecting Numerical Errors in Computation with Posits. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 731–746. doi:10.1145/3385412.3386004

[5] Sangeeta Chowdhary and Santosh Nagarakatte. 2022. Fast shadow execution for debugging numerical errors using error free transformations. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 190 (Oct. 2022), 28 pages. doi:10.1145/3563353

[6] H. J. Curnow and B. A. Wichmann. 1976. A synthetic benchmark. *Comput. J.* 19, 1 (01 1976), 43–49. doi:10.1093/comjnl/19.1.43

[7] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. 2017. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In *Numerical Software Verification*, Sergiy Bogomolov, Matthieu Martel, and Pavithra Prabhakar (Eds.). Springer International Publishing, Cham, 63–77. doi:10.1007/978-3-319-54292-8_6

[8] Eva Darulova and Viktor Kuncak. 2014. Sound Compilation of Reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 235–248. doi:10.1145/2535838.2535874

[9] Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. 2020. Scalable yet Rigorous Floating-Point Error Analysis. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–14. doi:10.1109/SC41405.2020.00055

[10] Nestor Demeure. 2021. *Gestion du compromis entre la performance et la précision de code de calcul.* Theses. Université Paris-Saclay. https://theses.hal.science/tel-03116750

[11] O. Flatt and P. Panchekha. 2023. Making Interval Arithmetic Robust to Overflow. In *2023 IEEE 30th Symposium on Computer Arithmetic (ARITH)*. IEEE Computer Society, Los Alamitos, CA, USA, 44–47. doi:10.1109/ARITH58626.2023.00022

[12] 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), 13–es. doi:10.1145/1236463.1236468

[13] Hui Guo and Cindy Rubio-González. 2018. Exploiting Community Structure for Floating-Point Precision Tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 333–343. doi:10.1145/3213846.3213862

[14] Ariel E. Kellison and Justin Hsu. 2024. Numerical Fuzz: A Type System for Rounding Error Analysis. *Proc. ACM Program. Lang.* 8, PLDI, Article 226 (June 2024), 25 pages. doi:10.1145/3656456

[15] Victor Magron, George Constantinides, and Alastair Donaldson. 2017. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Trans. Math. Softw.* 43, 4, Article 34 (Jan. 2017), 31 pages. doi:10.1145/3015465

[16] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. doi:10.1145/1273442.1250746

[17] Pavel Panchekha. 2021. *An Accurate Quadratic Formula.* Retrieved 16-March-2025 from https://pavpanchekha.com/blog/accurate-quadratic.html

[18] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/2737924.2737959

[19] Nathalie Revol and Fabrice Rouillier. 2005. Motivations for an Arbitrary Precision Interval Arithmetic and the MPFI Library. *Reliable Computing* 11, 4 (2005), 275–290. doi:10.1007/s11155-005-6891-y

[20] Brett Saiki, Jackson Brough, Jonas Regehr, Jesus Ponce, Varun Pradeep, Aditya Akhileshwaran, Zachary Tatlock, and Pavel Panchekha. 2025. Target-Aware Implementation of Real Expressions. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1069–1083. doi:10.1145/3669940.3707277

[21] B. Saiki, O. Flatt, C. Nandi, P. Panchekha, and Z. Tatlock. 2021. Combining Precision Tuning and Rewriting. In *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–8. doi:10.1109/ARITH51176.2021.00013

[22] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding root causes of floating point error. *SIGPLAN Not.* 53, 4 (June 2018), 256–269. doi:10.1145/3296979.3192411

[23] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

[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. doi:10.1145/3230733

[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*, Isil Dillig and Jens Palsberg (Eds.). Springer International Publishing, Cham, 516–537. doi:10.1007/978-3-319-73721-8_24