Detecting Floating-Point Errors via Atomic Conditions

DAMING ZOU, Peking University, China
MUHAN ZENG, Peking University, China
YINGFEI XIONG*, Peking University, China
ZHOULAI FU, IT University of Copenhagen, Denmark
LU ZHANG, Peking University, China
ZHENDONG SU, ETH Zurich, Switzerland

This paper tackles the important, difficult problem of detecting program inputs that trigger large floating-point errors in numerical code. It introduces a novel, principled *dynamic analysis* that leverages the mathematically rigorously analyzed *condition numbers* for atomic numerical operations, which we call *atomic conditions*, to effectively guide the search for large floating-point errors. Compared with existing approaches, our work based on atomic conditions has several distinctive benefits: (1) it does not rely on high-precision implementations to act as approximate oracles, which are difficult to obtain in general and computationally costly; and (2) atomic conditions provide accurate, modular search guidance. These benefits in combination lead to a highly effective approach that detects more significant errors in real-world code (*e.g.*, widely-used numerical library functions) and achieves several orders of speedups over the state-of-the-art, thus making error analysis significantly more practical. We expect the methodology and principles behind our approach to benefit other floating-point program analysis tasks such as debugging, repair and synthesis. To facilitate the reproduction of our work, we have made our implementation, evaluation data and results publicly available on GitHub at https://github.com/FP-Analysis/atomic-condition.

CCS Concepts: • **Software and its engineering → General programming languages**; **Software testing and debugging**.

Additional Key Words and Phrases: floating-point error, atomic condition, testing, dynamic analysis

ACM Reference Format:
Daming Zou, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, Lu Zhang, and Zhendong Su. 2020. Detecting Floating-Point Errors via Atomic Conditions. *Proc. ACM Program. Lang.* 4, POPL, Article 60 (January 2020), 27 pages. https://doi.org/10.1145/3371128

*Yingfei Xiong is the corresponding author.

Authors' addresses: Daming Zou, Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China, Department of Computer Science and Technology, Peking University, Beijing, 100871, China, zoudm@pku.edu.cn; Muhan Zeng, Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China, Department of Computer Science and Technology, Peking University, Beijing, 100871, China, mhzeng@pku.edu.cn; Yingfei Xiong, Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China, Department of Computer Science and Technology, Peking University, Beijing, 100871, China, xiongyf@pku.edu.cn; Zhoulai Fu, Department of Computer Science, IT University of Copenhagen, Rued Langgaards Vej 7, Copenhagen, 2300, Denmark, zhfu@itu.dk; Lu Zhang, Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China, Department of Computer Science and Technology, Peking University, Beijing, 100871, China, zhanglucs@pku.edu.cn; Zhendong Su, Department of Computer Science, ETH Zurich, Universitatstrasse 6, Zurich, 8092, Switzerland, zhendong.su@inf.ethz.ch.

2475-1421/2020/1-ART60
https://doi.org/10.1145/3371128

1 INTRODUCTION

Floating-point computation is important in science, engineering, and finance applications [Sanchez-Stern et al. 2018]. It is well-known that floating-point computation can be inaccurate due to the finite representation of floating-point numbers, and inaccuracies can lead to catastrophes, such as stock market disorder [Quinn 1983], incorrect election results [Weber-Wulff 1992], rocket launch failure [Lions et al. 1996], and the loss of human lives [Skeel 1992]. Modern systems also suffer from numerical inaccuracies, such as probabilistic programming systems [Dutta et al. 2018] and deep learning libraries [Pham et al. 2019]. The increased complexity of modern systems makes it even more important and challenging to detect floating-point errors.

Thus, it is critical to determine whether a floating-point program $\hat{f}$ has significant errors with respect to its mathematical oracle f, *i.e.*, an idealized implementation in the exact real arithmetic. This is a very challenging problem. As reported by Bao and Zhang [2013], only a small portion among all possible inputs would lead to significant errors in a program's final results. Several approaches [Chiang et al. 2014; Yi et al. 2017; Zou et al. 2015] have been proposed to detect inputs that trigger significant floating-point errors. All these approaches treat the floating-point program as a *black-box*, heavily depend on the oracle (using high precision program $\hat{f}_{high}$ to simulate the mathematical oracle f), and apply heuristic methods to detect significant errors.

However, it is expensive to obtain the simulated oracle $\hat{f}_{high}$ of an arbitrary program on an arbitrary input (Section 5.5.1). The cost is twofold. First, the high-precision program is computationally expensive — even programs with quadruple precision (128 bits) are 100x slower than double precision (64 bits) [Peter Larsson 2013]. The computational overhead further increases with higher precisions. Second, realizing a high-precision implementation is also expensive in terms of development cost as one cannot simply change all floating-point variables to high-precision types, but needs expert knowledge to handle *precision-specific operations* and *precision-related code* (*e.g.*, hard-coded series expansions and hard-coded iterations). More concretely, precision-specific operations are designed to work on a specific floating-point type [Wang et al. 2016], *e.g.*, operations only work on double precision (64 bits). Here is a simplified example from the `exp` function in the GNU C Library:

```
1  double round(double x) {
2    double n = 6755399441055744.0; // n equals to 3 << 51
3    return (x + n) - n; }
```

The above code snippet rounds x to an integer and returns the result. The constant n is a "magic number" and only works correctly for double precision (64 bits) floating-point numbers. The decimal digits of $(x + n)$ are rounded due to the limited precision, and the subtraction gives the result of rounding x to an integer. Changing the precision of floating-point numbers, regardless higher or lower, violates the semantics of precision-specific operations and leads to less accurate results [Wang et al. 2016]. Here is another example of precision-related code: the hard-coded series for calculating `sin(x)` near $x = 0$:

```
1  double sin(double x) {
2    if (x > -0.01 && x < 0.01) {
3      double y = x*x;
4      double c1 = -1.0 / 6.0;
5      double c2 =  1.0 / 120.0;
6      double c3 = -1.0 / 5040.0;
7      double sum = x*(1.0 + y*(c1 + y*(c2 + y*c3)));
8      return sum;
9    }
10   else { ... } }
```

The above code snippet calculates $\sin(x)$ for x near 0 based on the Taylor series of $\sin(x)$ at $x = 0$:

$$\sin(x) = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + O(x^8)$$

In this example of precision-related code, changing from floating-point to higher precision cannot make the result more accurate, since the error term $O(x^8)$ always exists. Due to the lack of automated tools, implementing a high quality high-precision oracle demands much expertise and effort, and thus is expensive in terms of development cost.

Recognizing the aforementioned challenges, we introduce an approach that is fundamentally different from existing black-box approaches. Our approach is based on analyzing and understanding how errors are introduced, propagated, and amplified during floating-point operations. In particular, we adapt the concept of *condition number* [Higham 2002] from numerical analysis to avoid estimating the oracle. Condition number measures how much the output value of the function can change for a small change in the input arguments. Importantly, we focus on condition numbers of a set of individual floating-point operations (such as $+$, $-$, $\sin(x)$, and $\log(x)$), which we term as *atomic condition*.

Our insight is that the atomic condition consists in a dominant factor for floating-point errors from atomic operations, which we can leverage to find large errors of a complex floating-point program. As such, we can express a floating-point error as $\varepsilon_{out} = \varepsilon_{in}\Gamma + \mu$ where Γ is the atomic condition and μ refers to an *introduced error* by atomic operations. The latter is guaranteed to be small by the IEEE 754 standard [Zuras et al. 2008] and the GNU C reference manual [Loosemore et al. 2019], because the atomic operations are carefully implemented and maintained by experts.

Based on this insight, we propose an approach based on atomic conditions and its realization, Atomu, for detecting floating-point errors. The approach critically benefits from our insight:

- **Native and Fast**: Atomic conditions can be computed with normal precision floating-point arithmetic, thus leading to high runtime efficiency.
- **Effective**: Atomic conditions provide accurate information on how errors are introduced and amplified by atomic operations. This information can be leveraged to effectively guide the search for error-triggering inputs.
- **Oracle-free**: Atomic conditions allow our approach to be independent of high-precision implementations (*i.e.*, oracles), thus making it generally applicable.
- **Easy-to-debug**: Atomic conditions help pinpoint where errors are significantly amplified by atomic operations, and thus the root causes of significant errors.

At the high level, Atomu searches within the whole floating-point space for significant atomic conditions. It returns a ranked list of test inputs that are highly likely to trigger large floating-point errors, which developers can review and confirm the existence of significant errors. Thus, developers only need to manually check a few inputs rather than attempting to construct a full oracle, which is both *much more expensive* to construct and run for finding error-triggering inputs. Furthermore, if oracles are available, our approach can leverage them and is fully automated end-to-end.

We evaluate Atomu on 88 functions from the popular GNU Scientific Library (GSL) to demonstrate the effectiveness and runtime efficiency of Atomu. We find that Atomu is at least two orders of magnitude faster than state-of-the-art approaches [Yi et al. 2019; Zou et al. 2015]. When oracles are available (*i.e.*, the same setting as all existing approaches), Atomu can detect significantly more (40%) buggy functions with significant errors *with neither false positives nor false negatives on real-world evaluation subjects* (see Section 5.3.2). For cases where oracles do not exist, none of the state-of-the-art approaches is applicable, but Atomu can accurately identify 74% of buggy functions by checking only the top-1 generated inputs and 95% by checking only the top-4 generated inputs.

In summary, we make the following main contributions in this paper:

- We introduce the concept of atomic conditions and use it to explain how floating-point errors are introduced, propagated, and amplified by atomic operations;
- We introduce and formulate the insight that atomic conditions are the dominant factors for floating-point errors, and analyze the atomic conditions for a realistic collection of operations, such as $x + y$, $\sin(x)$, $\log(x)$, and $\text{sqrt}(x)$;
- We design and realize Atomu based on the insight of atomic conditions. In particular, Atomu detects significant atomic conditions, which are highly related to significant errors in the results. As Atomu does not rely on oracles, it is both generally applicable and efficient; and
- We extensively evaluate Atomu on functions from the widely-used GSL, and evaluation results demonstrate that Atomu is orders of magnitude faster than the state-of-the-art and significantly more effective — it reports more buggy functions (which are missed by state-of-the-art approaches), and incurs neither false positives nor false negatives.

2 PRELIMINARIES

This section presents the necessary background on floating-point representations, error measurement and analysis, errors in atomic operations, and their condition numbers.

2.1 Floating-Point Representation

Floating-point numbers approximate real numbers, and can only represent a finite subset of the continuum of real numbers. According to the IEEE 754 standard [Zuras et al. 2008], the representations of floating-point numbers consist of three parts: sign, exponent, and significand (also called mantissa). Table 1 shows the format of the three parts in half, single, and double precision floating-point numbers.

Table 1. IEEE 754 floating-point representation.

	Sign	Exponent (m)	Significand (n)
Half Precision (16 bits)	1	5	10
Single Precision (32 bits)	1	8	23
Double Precision (64 bits)	1	11	52

The value of a floating-point number is inferred by the following rules: If all bits in the exponent are 1, the floating-point representation denotes one of several special values: $+\infty$, $-\infty$, or *NaN (not-a-number)*. Otherwise, the value of a floating-point representation is depicted by

$$(-1)^S \times T \times 2^E, \text{where}$$

- S is the value of the sign bit, 0 for positive and 1 for negative;
- $E = e - bias$, where e is the biased exponent value with m bits, and $bias = 2^{m-1} - 1$;
- $T = d_0.d_1d_2 \ldots d_n$, where $d_1d_2 \ldots d_n$ are the n bits trailing significand field, and the leading bit d_0 is implicitly encoded in the biased exponent e.

2.2 Error Measurement

Since floating-point numbers ($\mathbb{F}$) use finite bits to represent real numbers, they are inevitably inaccurate for most real numbers. One consequence of the inaccuracy is *rounding errors*. We represent the error between a real number x and a floating-point number $\hat{x}$ as $x = \hat{x} + \eta$, where η denotes the rounding error due to insufficient precision in the floating-point representation.

For a floating-point program $\mathbf{P}$: $\hat{y} = \hat{f}(\mathbf{x})$, $\hat{y} \in \mathbb{F}$, rounding errors will be introduced and accumulated for each floating-point operation, and the accumulated errors during the whole

computation may lead to inaccurate output results. There are two standard ways to measure the error between the ideal mathematical result $f(\mathbf{x})$ and the floating-point program result $\hat{f}(\mathbf{x})$: *absolute error* $Err_{abs}(f(\mathbf{x}), \hat{f}(\mathbf{x}))$ and *relative error* $Err_{rel}(f(\mathbf{x}), \hat{f}(\mathbf{x}))$, defined respectively as:

$$Err_{abs}(f(\mathbf{x}), \hat{f}(\mathbf{x})) = \left| f(\mathbf{x}) - \hat{f}(\mathbf{x}) \right| \qquad Err_{rel}(f(\mathbf{x}), \hat{f}(\mathbf{x})) = \left| \frac{f(\mathbf{x}) - \hat{f}(\mathbf{x})}{f(\mathbf{x})} \right|$$

Units in the last place (ULP) is a measure for the relative error [Loosemore et al. 2019; Zuras et al. 2008]. For a real number z with the floating-point represented by floating-point as $z = (-1)^S \times d_0.d_1 \ldots d_n \times 2^E$, ULP and the error based on ULP are represented by:

$$ULP(z) = \frac{\left| d_0.d_1 \ldots d_n - (z/2^E) \right|}{2^{n-1}} \qquad Err_{ulp}(f(\mathbf{x}), \hat{f}(\mathbf{x})) = \left| \frac{f(\mathbf{x}) - \hat{f}(\mathbf{x}))}{ULP(f(\mathbf{x}))} \right|$$

For double precision (64-bits) floating-point numbers, 1 ULP error (Err_{ulp}) corresponds to a relative error between 1.1×10^{-16} and 2.2×10^{-16}. For many applications this error is quite small, *e.g.*, considering the radius of our solar system, which is about 4.503 billion kilometers from the Sun to Neptune, an ULP error of this distance is less than 1 millimeter.

Following the common practice [Goldberg 1991; Higham 2002], relative error Err_{rel} is the prevalent measurement for floating-point errors.

2.3 Errors in Floating-Point Atomic Operations

A floating-point program's computation consists of a series of *atomic operations*, which include the following elementary arithmetic and basic functions:

- Elementary arithmetic: $+$, $-$, $\times$, $\div$.
- Basic functions:
  - Trigonometric functions: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`;
  - Hyperbolic functions: `sinh`, `cosh`, `tanh`;
  - Exponential and logarithmic functions: `exp`, `log`, `log10`; and
  - Power functions: `sqrt`, `pow`.

According to the IEEE 754 standard [Zuras et al. 2008], elementary arithmetic operations are guaranteed to produce accurately rounded results and their errors never exceed 1+1/1000 ULPs.

As for the basic functions, according to the GNU C Library reference manual [Loosemore et al. 2019], *"Ideally the error for all functions is always less than 0.5 ULPs in round-to-nearest mode."* This manual also provides a list of known maximum errors for math functions. By checking the list, the maximum errors in the aforementioned basic functions are 2 ULPs on the x86_64 architecture.

In summary, these two definitive documents stipulate: *An atomic operation is guaranteed to be accurate. The error introduced by atomic operations is usually less than 0.5 ULP and at most 2 ULPs.*

It is also well-known that floating-point programs can be significantly inaccurate on specific inputs [Panchekha et al. 2015; Zou et al. 2015]. For some functions from well-maintained and widely-deployed numerical library, GNU Scientific Library, their relative errors may be larger than 0.1, corresponding to 7×10^{14} ULPs. Such large errors occur because, during the computation, errors are not only introduced and accumulated, but also *amplified* by certain operations.

2.4 Condition Numbers

The *Condition number* is an important quantity in numerical analysis. It measures the inherent stability of a mathematical function f and is independent of the implementation $\hat{f}$ [Higham 2002].

Assuming an input x carries a small error Δx, the following equation measures how much the error Δx will be *amplified* by the mathematical function f. Here we assume for simplicity that f is

twice continuously differentiable:

$$\begin{aligned} Err_{rel}(f(x), f(x+\Delta x)) &= \left|\frac{f(x+\Delta x)-f(x)}{f(x)}\right| \\ &= \left|\frac{f(x+\Delta x)-f(x)}{\Delta x}\cdot\frac{\Delta x}{f(x)}\right| \\ &= \left|(f'(x)+\frac{f''(x+\theta\Delta x)}{2!}\Delta x)\cdot\frac{\Delta x}{f(x)}\right|, \theta\in(0,1) \\ &= \left|\frac{\Delta x}{x}\right|\cdot\left|\frac{xf'(x)}{f(x)}\right| + O\big((\Delta x)^2\big) \\ &= Err_{rel}(x, x+\Delta x)\cdot\left|\frac{xf'(x)}{f(x)}\right| + O\big((\Delta x)^2\big) \end{aligned}$$

where θ denotes a value $\in (0,1)$ and is the Lagrange form of the remainder in the Taylor Theorem [Kline 1998]. This equation leads to the notion of a condition number [Higham 2002]:

$$\Gamma_f(x) = \left|\frac{xf'(x)}{f(x)}\right|$$

The condition number measures the relative change in the output for a given relative change in the input, *i.e.*, how much the relative error $|\Delta x/x|$ carried by the input will be amplified in the output by the mathematical function f.

Note that computing the condition number $\Gamma_f(x)$ is commonly regarded as more difficult than computing the original mathematical function $f(x)$ [Higham 2002], since $f'(x)$ cannot be easily obtained unless some quantities are already pre-calculated [Fu et al. 2015].

3 EXAMPLE

This section uses a concrete example to motivate and illustrate our approach. Via the given example, we show how atomic operations affect the accuracies of results and how we use atomic conditions to uncover and diagnose floating-point errors.

Let us consider a numerical program $\hat{f}$: foo(x) for calculating the value of a mathematical function f defined in Figure 1. And the pseudocode for program $\hat{f}$: foo(x) is listed in the first column of the table in Figure 1. Note that the limit of $f(x)$, as x approaches 0, is $1/2$.

Naturally, the programmer would expect the program $\hat{f}$: foo(x) to produce an accurate result. However, when the input is 10^{-7}, which is a small number close to 0, the result of $\hat{f}$ becomes significantly inaccurate, *i.e.*, $\hat{f}(10^{-7}) \approx 0.4996$. The accurate result of f, which we use the high-precision program $\hat{f}_{high}$ to simulate, is $\hat{f}_{high}(10^{-7}) = 0.499999999999999583$.

To illustrate how errors are introduced and amplified by atomic operations, let us inspect the intermediate variables one by one. Figure 1 shows the operands, atomic conditions, results, and the relative errors of the four operations. The inaccurate digits in the operands and results are highlighted in bold (*e.g.*, `4.99`**`6003610813204 43`**`e-15`). Each operation (1) introduces a rounding error around 1 ULP as discussed in Section 2.3, and (2) amplifies the existing error in operand(s) with a factor of its atomic condition as discussed in Section 2.4.

Note that $\hat{f}_{high}$ is only used to calculate the relative error; computing the result and atomic condition does not need the high-precision program.

- **op 1: `v1 = cos(x)`**.
  - *Atomic condition formula*: $\Gamma_{\cos}(x) = |x \cdot \tan(x)|$.

Example Function: $f(x) = \dfrac{1-\cos(x)}{x^2}$ $\quad\quad$ $\lim\limits_{x\to 0} f(x) = \dfrac{1}{2}$

Pseudocode of $\hat{f}$: `foo(x)`	**Operand(s)**	**Atomic condition Γ_{op}**	**Operation result in $\hat{f}$**	**Relative error in operation result**
`foo(double x):`				
`v1=cos(x)`	`1.0e-7`	1.0000e-14	`9.9999999999995004e-01`	3.9964e-18
`v2=1.0-v1`	`1.0`, `9.9999999999995004e-01`	2.0016e+14, 2.0016e+14	`4.996003610813204`**`43`**`e-15`	7.9928e-04
`v3=x*x`	`1.0e-7`, `1.0e-7`	1, 1	`9.999999999999984`**`1`**`e-15`	6.8449e-17
`v4=v2/v3`	`4.996003610813204`**`43`**`e-15`, `9.999999999999984`**`1`**`e-15`	1, 1	`4.996003610813204`**`99`**`e-01`	7.9928e-04
`return v4`				

Fig. 1. An Example of mathematical function f and error propagation in atomic operations of $\hat{f}$: foo(x).

  - *Amplified error*: Since x, the input to the program, is treated as error-free [Loosemore et al. 2019], this operation will not amplify the error in its operand by atomic condition.
  - *Introduced error*: 3.9964×10^{-18}, which is smaller than 1 ULP.
- **op 2: `v2 = 1.0 - v1`**.
  - *Atomic condition formula*: $\Gamma_{-}(v1) = \left|-\frac{v1}{1.0-v1}\right| = 2.0016 \times 10^{14}$.
  - *Amplified error*: The operand v1 contains a quite small relative error 3.9964×10^{-18}. However, since the atomic condition is very large, this relative error will be amplified to 7.9928×10^{-4} in the result.
  - *Introduced error*: Around 1 ULP, which can be omitted due to the large amplified error.
- **op 3: `v3 = x * x`**.
  - *Atomic condition formula*: $\Gamma_{\times}(x) = 1$. The atomic condition of multiplication is always equal to 1, which means that the error in its operands will simply pass to the result without being amplified or reduced.
  - *Amplified error*: Since x is error-free, there is no amplified error in the result.
  - *Introduced error*: 6.8449×10^{-17}, which is smaller than 1 ULP.
- **op 4: `v4 = v2 / v3`**.
  - *Atomic condition formula*: $\Gamma_{\div}(v2) = \Gamma_{\div}(v3) = 1$. The atomic condition of division is always 1.
  - *Amplified error*: The existing error in v2 is passed (amplified by 1) to the result as a relative error 7.9928×10^{-4}. The amplified error from v3 is much smaller and can be omitted.
  - *Introduced error*: Around 1 ULP, which can be omitted due to the large amplified error.

This example illustrates that

- A significant error in the result is mostly caused by one or multiple significant amplifications by atomic operations;
- Atomic condition can reveal/measure whether an atomic operation will lead to significant amplifications; and
- Computing atomic conditions is low-cost, since the formulae for atomic conditions can be pre-calculated, and there is no need for high-precision implementations.

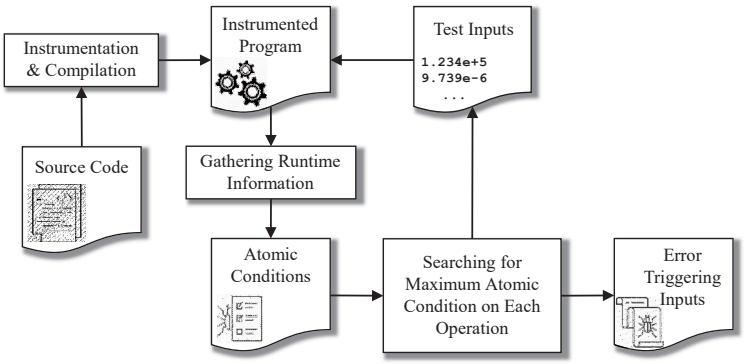


Fig. 2. Overview of atomic condition-driven error analysis.

4 ERROR ANALYSIS VIA ATOMIC CONDITIONS

This section presents the technical details of our approach. We start by formulating the problem and giving an overview of our approach (Section 4.1). Next, we analyze the propagation of errors via atomic conditions, describe the pre-calculated formulae for atomic conditions and introduce the notion of *danger zones* (Section 4.2). We then discuss our search framework and algorithm (Section 4.3) and how to rank/prioritize results (Section 4.4).

Section 4.2 provides the theoretical underpinning for using atomic conditions to indicate significant floating-point errors, while Section 4.3 and Section 4.4 concern the practical aspects of using atomic conditions for detecting such errors.

4.1 Problem Definition and Approach Overview

Given a floating-point program $\mathbf{P}$: $\hat{y} = \hat{f}(\mathbf{x})$, $\hat{y} \in \mathbb{F}$, our goal is to find a set of inputs $\mathbf{x}$ to $\mathbf{P}$ that leads to large floating-point errors, which we cast as a search problem. The search space is represented by all possible values of $\mathbf{x}$ within the domain of $\mathbf{P}$. The domain of $\mathbf{P}$ could either be *explicit*, such as described in its documentation, or *implicit*, which may only be inferred through exceptions or status variables.

For the search problem, we need to specify the criterion which critically dictates what inputs are of interest and how the search can be guided. For our problem setting, the criterion of an input $\mathbf{x}$ in the search space should determine how likely $\mathbf{x}$ is to lead a significant error. A natural method to define the criterion is to directly use the relative error $Err_{rel}(f(\mathbf{x}), \hat{f}(\mathbf{x}))$ [Zou et al. 2015]. However, as discussed in Section 5.5.1, $f(\mathbf{x})$ is only conceptual and the high-precision results $\hat{f}_{high}(\mathbf{x})$ are needed to simulate/approximate $f(\mathbf{x})$. There are technical and practical disadvantages of adopting a high-precision $\hat{f}_{high}$ (Section 1). Thus, an alternative criterion is needed to guide our search procedure, which Section 4.2 motivates and introduces.

Figure 2 shows a high-level overview of our approach. It instruments the source code of a floating-point program to emit runtime information, which is used to compute the atomic condition on each atomic operation. The search algorithm iteratively generates inputs and computes the atomic conditions to find inputs that trigger large atomic conditions on each atomic operation.

4.2 Error Propagation, Atomic Conditions and Danger Zones

We analyze the program $\mathbf{P}$ in a white-box manner, where we assume its source code is available. The program consists of atomic operations as defined in Section 2.3. We define and call the condition

number (Section 2.4) of an atomic operation as its *atomic condition*. For brevity of exposition, we consider a univariate operation $z = op(x)$:

- x is the input, which carries a relative error ε_x,
- $\Gamma_{op}(x)$ is the atomic condition of $op(x)$,
- z is the output, which carries an unknown relative error ε_z,
- $\mu_{op}(x)$ is the introduced error as discussed in Section 2.3.

Based on the discussion in Section 2.4, the atomic operation op amplifies the error ε_x with a factor of atomic condition $\Gamma_{op}(x)$, and the amplified error is thus $\varepsilon_x\Gamma_{op}(x)$. Also as discussed in Section 2.3, the atomic operation also introduces a small error around 1 ULP, which is $\mu_{op}(x)$. The error ε_z can be presented as:

$$\varepsilon_z = \varepsilon_x\Gamma_{op}(x) + \mu_{op}(x) \tag{1}$$

Equation (1) can be easily generalized to multivariate operations $z = op(x, y)$ such as $z = x + y$:

$$\varepsilon_z = \varepsilon_x\Gamma_{op,x}(x, y) + \varepsilon_y\Gamma_{op,y}(x, y) + \mu_{op}(x, y) \tag{2}$$

where $\Gamma_{op,x}(x, y)$ and $\Gamma_{op,y}(x, y)$ are based on the partial derivatives with respect to x and y.

We first use a small example to demonstrate the propagation model based on Equations (1) and (2). Then, we discuss the generalized propagation model. Consider the following function `bar`:

```
1  double bar(double x) {
2    double v1,v2,v3; // intermediate variables
3    double y;        // return value
4    v1 = f1(x);
5    v2 = f2(v1);
6    v3 = f3(v1,v2);
7    y = f4(v3);
8    return y; }
```

We assume that the four function invocations, namely f1, f2, f3, and f4, are atomic operations (*e.g.*, log, sin, +, sqrt, *etc.*). Following Equations (1) and (2), we have the following equations, where, for simplicity, the parameters of Γ_{op} and μ_{op} are implicit:

$$\begin{aligned}
\varepsilon_x &= \mu_{init} \\
\varepsilon_{v1} &= \varepsilon_x\Gamma_{f_1} + \mu_{f_1} \\
\varepsilon_{v2} &= \varepsilon_{v1}\Gamma_{f_2} + \mu_{f_2} \\
\varepsilon_{v3} &= \varepsilon_{v1}\Gamma_{f_3,v1} + \varepsilon_{v2}\Gamma_{f_3,v2} + \mu_{f_3} \\
\varepsilon_y &= \varepsilon_{v3}\Gamma_{f_4} + \mu_{f_4}
\end{aligned}$$

After expansion, we obtain the following:

$$\begin{aligned}
\varepsilon_y = {} & \overbrace{\mu_{init}\Gamma_{f_1}\Gamma_{f_3,v1}\Gamma_{f_4}}^{x\to v1\to v3\to y} + \overbrace{\mu_{init}\Gamma_{f_1}\Gamma_{f_2}\Gamma_{f_3,v2}\Gamma_{f_4}}^{x\to v1\to v2\to v3\to y} \\
& + \overbrace{\mu_{f_1}\Gamma_{f_3,v1}\Gamma_{f_4}}^{v1\to v3\to y} + \overbrace{\mu_{f_1}\Gamma_{f_2}\Gamma_{f_3,v2}\Gamma_{f_4}}^{v1\to v2\to v3\to y} \\
& + \overbrace{\mu_{f_2}\Gamma_{f_3,v2}\Gamma_{f_4}}^{v2\to v3\to y} + \overbrace{\mu_{f_3}\Gamma_{f_4}}^{v3\to y} + \overbrace{\mu_{f_4}}^{y}
\end{aligned}$$

From the above equation, we observe that each of the introduced error terms μ is amplified by the atomic condition Γ_{op} through a data-flow path to the result. For example, there are two data-flow

paths from x to y: (1) $x \to v1 \to v3 \to y$, corresponding to the first term of the equation; and (2) $x \to v1 \to v2 \to v3 \to y$, corresponding to the second term of the equation.

More generally, with the following definitions and notations

- P is a floating-point program.
- E is a set of atomic operations op in P, as edges;
- V is a set of floating-point variables v in P, as vertices;
- $G : \langle V, E \rangle$ is a dynamic data-flow graph with entry vertex x and exit vertex y. For an executed atomic operation $\gamma = op(\alpha)$, there is an edge $op : \alpha \to \gamma$. For an executed atomic operation $\gamma = op(\alpha, \beta)$, there are two edges $op_\alpha : \alpha \to \gamma$ and $op_\beta : \beta \to \gamma$;
- $m = [\alpha, \beta, \ldots, y]$ is an executed data-flow path in G from variable α to result variable y. Note that there may exist multiple paths from the same variable α to y;
- $M(\alpha)$ is the set of all executed data-flow paths from α to y; and
- $res(op) : op \to \gamma$ is the mapping from an atomic operation op to its result γ.

The propagation model of error ε_y can be formalized as

$$\varepsilon_y = \sum_{e \in E} \left(\mu_e \cdot \sum_{m \in M(res(e))} \prod_{op \in m} \Gamma_{op} \right) \tag{3}$$

Equation (3) highlights our key insight: The error in result ε_y is determined only by the atomic condition Γ_{op} and the introduced error μ_e. Since the introduced error is guaranteed to be small (at most 2 ULPs, Section 2.3), atomic conditions are the dominant factors of floating-point errors.

As we discussed in Section 2.4, computing the condition number $\Gamma_f(x)$ is commonly regarded as more difficult than computing the original function $f(x)$ [Higham 2002], since $f'(x)$ cannot be easily obtained unless certain quantities have already been pre-calculated [Fu et al. 2015].

Rather, we focus on the atomic operations, all of which are basic mathematical functions. These atomic operations are all twice continuously differentiable, and their derivatives have analytic expressions. Thus, we can use pre-calculated formulae to compute the atomic conditions, which reduces computational effort.

Table 2 lists the atomic condition formulae for all atomic operations described in Section 2.3. We define the *danger zone* to be the domain for x (or y) that triggers a significantly large atomic condition. By analyzing the maximal values of the atomic condition formulae, we classify the operations into two categories:

- **Potentially unstable operations:** $+$, $-$, sin, cos, tan, arcsin, arccos, sinh[1], cosh[1], exp[1], pow[2], log, log10. For each of these operations, if its operator falls into its danger zone, the atomic condition becomes significantly large ($\Gamma_{op} \to +\infty$), and any slight error in the operator will be amplified to a tremendous inaccuracy in the operation's result.
- **Stable operations:** $\times$, $\div$, arctan, arctan2, tanh, sqrt. For each of these operations, its atomic condition is always no greater than 1, which means that the error in the operator will not be amplified in the operation's result. This is determined by the mathematical formulae of atomic conditions. For example, consider the atomic condition of $\tanh(x)$:

$$\Gamma_{\tanh}(x) = \begin{cases} \left| \frac{x}{\sinh(x)\cosh(x)} \right| < 1 & \text{if } x \in \mathbb{R} : x \neq 0, \\ \lim_{x \to 0} \left| \frac{x}{\sinh(x)\cosh(x)} \right| = 1 & \text{if } x = 0. \end{cases}$$

[1]For a 64-bit floating-point program, the domain (without triggering overflow or underflow) is (-709.78, 709.78) for $\exp(x)$, and (-710.47, 710.47) for $\sinh(x)$ and $\cosh(x)$. The range of domain restricts the atomic condition $\Gamma_{\exp}$, $\Gamma_{\sinh}$, $\Gamma_{\cosh}$ from becoming extremely large as other potentially unstable operations.

[2]The power function x^y is similar with the $\exp(x)$ function that the domain for y is limited. The condition of this function has been discussed in previous work [Harrison 2009].

Table 2. Pre-calculated atomic condition formulae.

Operation (op)	Atomic Condition (Γ_{op})	Danger Zone
$op(x,y) = x + y$	$\Gamma_{+,x}(x,y) = \left\lvert\frac{x}{x+y}\right\rvert$, $\Gamma_{+,y}(x,y) = \left\lvert\frac{y}{x+y}\right\rvert$	$x \approx -y$
$op(x,y) = x - y$	$\Gamma_{-,x}(x,y) = \left\lvert\frac{x}{x-y}\right\rvert$, $\Gamma_{-,y}(x,y) = \left\lvert-\frac{y}{x-y}\right\rvert$	$x \approx y$
$op(x,y) = x \times y$	$\Gamma_{\times,x}(x,y) = \Gamma_{\times,y}(x,y) = 1$	-
$op(x,y) = x \div y$	$\Gamma_{\div,x}(x,y) = \Gamma_{\div,y}(x,y) = 1$	-
$op(x) = \sin(x)$	$\Gamma_{\sin}(x) = \lvert x \cdot \cot(x)\rvert$	$x \to n\pi, n \in \mathbb{Z}$
$op(x) = \cos(x)$	$\Gamma_{\cos}(x) = \lvert x \cdot \tan(x)\rvert$	$x \to n\pi + \frac{\pi}{2}, n \in \mathbb{Z}$
$op(x) = \tan(x)$	$\Gamma_{\tan}(x) = \left\lvert\frac{x}{\sin(x)\cos(x)}\right\rvert$	$x \to \frac{n\pi}{2}, n \in \mathbb{Z}$
$op(x) = \arcsin(x)$	$\Gamma_{\arcsin}(x) = \left\lvert\frac{x}{\sqrt{1-x^2}\cdot\arcsin(x)}\right\rvert$	$x \to -1^+, x \to 1^-$
$op(x) = \arccos(x)$	$\Gamma_{\arccos}(x) = \left\lvert-\frac{x}{\sqrt{1-x^2}\cdot\arccos(x)}\right\rvert$	$x \to -1^+, x \to 1^-$
$op(x) = \arctan(x)$	$\Gamma_{\arctan}(x) = \left\lvert\frac{x}{(x^2+1)\cdot\arctan(x)}\right\rvert$	-
$op(x,y) = \arctan(\frac{y}{x})$	$\Gamma_{atan2,x}(x,y) = \Gamma_{atan2,y}(x,y) = \left\lvert\frac{xy}{(x^2+y^2)\arctan(\frac{y}{x})}\right\rvert$	-
$op(x) = \sinh(x)$	$\Gamma_{\sinh}(x) = \lvert x \cdot \coth(x)\rvert$	$x \to \pm\infty$
$op(x) = \cosh(x)$	$\Gamma_{\cosh}(x) = \lvert x \cdot \tanh(x)\rvert$	$x \to \pm\infty$
$op(x) = \tanh(x)$	$\Gamma_{\tanh}(x) = \left\lvert\frac{x}{\sinh(x)\cosh(x)}\right\rvert$	-
$op(x) = \exp(x)$	$\Gamma_{\exp}(x) = \lvert x\rvert$	$x \to \pm\infty$
$op(x) = \log(x)$	$\Gamma_{\log}(x) = \left\lvert\frac{1}{\log x}\right\rvert$	$x \to 1$
$op(x) = \log_{10}(x)$	$\Gamma_{log10}(x) = \left\lvert\frac{1}{\log x}\right\rvert$	$x \to 1$
$op(x) = \sqrt{x}$	$\Gamma_{sqrt}(x) = 0.5$	-
$op(x,y) = x^y$	$\Gamma_{pow,x}(x,y) = \lvert y\rvert$, $\Gamma_{pow,y}(x,y) = \lvert y \log(x)\rvert$	$x \to 0^+, y \to \pm\infty$

The range of $\Gamma_{\tanh}(x)$ is $\{\Gamma_{\tanh} \in \mathbb{R} : 0 < \Gamma_{\tanh} \leq 1\}$. And the atomic conditions of other stable operations are also not greater than 1.

4.3 Atomic Condition-Guided Search

As stated earlier, we adopt a search-based approach, and the aim is to find inputs that trigger the largest atomic condition on each atomic operation. This section describes the technical details of our search algorithm. It is designed as a pluggable component of our overall approach, *i.e.*, other search algorithms could also be easily adopted. Our search algorithm operates as follows:

(1) Generate a set of initial test inputs;
(2) Invoke the program under analysis with the test inputs;
(3) Gather the atomic conditions on the atomic operations;
(4) Generate new test inputs based on the computed atomic conditions;
(5) Repeat steps (2) to (4) until a termination criterion is reached; and
(6) Report largest atomic condition found for each atomic operation and corresponding input.

In this paper, we propose an evolutionary algorithm (EA) for realizing the search module. Evolutionary algorithms [Bäck et al. 1999] simulate the process of natural selection for solving optimization problems. In EA, each candidate solution is called an individual, and there is a fitness function that determines the quality of the solutions. In our search module, the individual is defined

as a floating-point test input, and the fitness function is defined as the atomic condition on an atomic operation.

Algorithm 1: EvolutionSearch

Input: An instrumented floating-point program $\mathbb{P}$, the size of initialization *initSize*, the size of iterations on each operation *iterSize*

Output: A set of test inputs $X = \{x_1, x_2, \ldots, x_n\}$, corresponding to unstable operations $\{op_1, op_2, \ldots, op_n\}$

```
1  X ← ∅
2  initTests ← generateInitTests(initSize)
3  for test in initTests do
4      computeAllAtomicConditions(test)
5  for op_i in ℙ do
6      if isPotentialUnstable(op_i) then
7          T_i ← initTests
8          for j ← 0 to iterSize do
9              x ← selectTest(T_i, op_i)
10             x' ← mutateTest(x, j)
11             ac' ← computeAtomicCondition(op_i, x')
12             T_i.append({x', ac'})
13         {x_i, ac_i} ← bestOf(T_i)
14         if ac_i > unstableThreshold then
15             X.append(x_i)
16 return X
```

A high-level outline of our evolutionary algorithm is shown in Algorithm 1. There are three main components of the proposed algorithm: *initialization*, *selection*, and *mutation*. Next, we explain the details of these components.

Initialization. First, the algorithm generates, uniformly at random in the space of floating-point numbers, a set of floating-point inputs as candidates (line 2). Then, it executes the instrumented program on each input and records the atomic conditions on all the executed operations (lines 3-4). The atomic conditions are used in the subsequent steps.

As mentioned in Section 4.2, the atomic operations can be classified into two categories: potentially unstable operations and stable operations. Our search algorithm iteratively focuses on each potentially unstable operation (op_i) (lines 5-6), and searches for inputs that maximize the atomic condition on op_i (lines 8-12).

During each iteration, the algorithm selects a test input (line 9), mutates it (line 10), computes the atomic condition Γ_{op_i} (line 11), and puts it back to the pool of test inputs (line 12). After the iterations on op_i, the algorithm checks whether the largest atomic condition Γ_{op_i} exceeds the unstable threshold (line 13-14). If yes, the corresponding input x_i is added to the result set (line 15). Finally, after looping over all potentially unstable operations, the result set X is returned (line 16). Suppose that X contains n test inputs $\{x_1, x_2, \ldots, x_n\}$. Each x_i corresponds to an unstable operation op_i, *i.e.*, the atomic condition Γ_{op_i} exceeds the unstable threshold on the execution with input x_i.

Selection. This component is one of the main steps in evolutionary algorithms. Its primary objective is to favor good test inputs among all candidate inputs. We utilize a rank-based selection

method [Baker 1985], which sorts candidates based on their fitness values (*i.e.*, atomic conditions for our approach).

Next, we assign a selection probability to each candidate based on its rank. To this end, we use a geometric distribution to assign probabilities [Whitley 1989]. Suppose that a test input t_r has the r-th largest atomic condition on op_i over all M candidates, *e.g.*, the rank of the best candidate is 1, the probability of selecting t_r is defined as

$$P(t_r) = \frac{\alpha(1-\alpha)^{r-1}}{\sum_{j=1}^{M} \alpha(1-\alpha)^{j-1}}$$

where α is the parameter of the geometric distribution. This distribution arises if the selection acts as a result of independent Bernoulli trials over the test inputs in rank order. For example, assuming that $M = 10, \alpha = 0.2$, we have numerators $20\%, 16\%, 12.8\%, \ldots, 2.7\%$ following the geometric distribution, and then the probabilities are normalized to $22.4\%, 17.9\%, 14.3\%, \ldots, 3\%$ to satisfy $\sum(P(t_r)) = 1$.

Mutation. This is another main step in evolutionary algorithms. As the selection step favors good existing candidates, the mutation step generates new candidates based on the selected ones. The objective of mutation is to search among the existing good candidates to obtain a better one. Our approach adopts a common mutation method, which adds a standard Gaussian random variable to the selected candidate t [Bäck et al. 1999]

$$t' = t + t \cdot \mathcal{N}(0, \sigma_j)$$

where j is the iteration counter and σ_j the standard deviation of the Gaussian distribution. The parameter σ_j controls the search size of t's neighbor. For example, assuming that $t = 1.2$, t' may be mutated to 1.3 on large σ_j, and may be mutated to 1.2003 on small σ_j. To make the mutation fine-grained and adaptable, the component σ_j is tapered off during the iterations

$$\sigma_j = \sigma_{st}^{(N-j)/N} \cdot \sigma_{end}^{j/N}$$

where N is the number of total iterations, σ_{st} and σ_{end} two parameters. For example, assuming that $\sigma_{st} = 10^{-1}, \sigma_{end} = 10^{-7}$, and $N = 100$, we have $\sigma_0 = 10^{-1}$ in the first iteration, $\sigma_{50} = 10^{-4}$ in the median iteration, and $\sigma_{100} = 10^{-7}$ in the final iteration.

4.4 Input Ranking

Our search algorithm returns a set of inputs $X = \{x_1, x_2, \ldots, x_n\}$ that trigger large atomic conditions. However, although likely, it is not guaranteed that they lead to large relative errors. If a high-quality high-precision program $\hat{f}_{high}$ exists, it can act as the oracle to help compute the relative errors on these inputs. Since the number of returned inputs is typically small, validating these inputs with $\hat{f}_{high}$ is computationally cheap.

On the other hand, as discussed earlier, $\hat{f}_{high}$ is often unavailable. We thus propose a method to prioritize the returned inputs, in particular, to rank the most suspicious inputs highly. Let us recall the example error propagation in Section 4.2. The equation of the error in result (ε_y) is

$$\begin{aligned}\varepsilon_y =& \mu_{init}\Gamma_{f_1}\Gamma_{f_3,v1}\Gamma_{f_4} + \mu_{init}\Gamma_{f_1}\Gamma_{f_2}\Gamma_{f_3,v2}\Gamma_{f_4} \\ &+ \mu_{f_1}\Gamma_{f_3,v1}\Gamma_{f_4} + \mu_{f_1}\Gamma_{f_2}\Gamma_{f_3,v2}\Gamma_{f_4} \\ &+ \mu_{f_2}\Gamma_{f_3,v2}\Gamma_{f_4} + \mu_{f_3}\Gamma_{f_4} + \mu_{f_4}\end{aligned}$$

From the above equation, we can observe that the atomic conditions in the latter operations have more dominance on the final result. For example, Γ_{f_4} shows in every term except the last term μ_{f_4}, which means a significantly large Γ_{f_4} is more likely to lead to a significant error ε_y. It also can

be explained based on our general model in Equation (3): the latter operations are contained in more data-flow paths, leading to more dominance on the error ε_y.

Thus, we propose an intuitive, effective method to prioritize the results. For a test input x_i and its corresponding unstable operation op_i, $step_i$ is computed as the number of floating-point operations from op_i to the return statement in the execution of x_i. Then, we prioritize the test inputs $\{x_1, x_2, \ldots, x_n\}$ — the smaller $step_i$, the higher rank of x_i.

5 EVALUATION

This section details the realization and evaluation of our technique. We show that our approach drastically outperforms the state-of-the-art on real-world code: (1) It is effective and precise — it detects more functions with large floating-point errors without false positives nor false negatives; and (2) it is highly scalable — it is several orders of magnitude faster than the state-of-the-art.

5.1 Implementation

We realize our approach as the tool Atomu. It instruments a given program for obtaining its runtime information to help compute the atomic conditions. The instrumentation is done in three steps (assuming that the program under analysis is `sample.c`):

(1) *Source code → LLVM IR:* This step compiles the floating-point program `sample.c` to the LLVM Intermediate Representation (IR) [Lattner and Adve 2004]. For this purpose, we use Clang[3] for C/C++ programs;
(2) *Instrument LLVM IR:* This step of Atomu is implemented as an LLVM pass to perform the needed instrumentation. It scans the LLVM IR for `sample.c` instruction by instruction. Once it encounters one of the floating-point atomic operations, it injects a function call to an external handler by passing the type of the operation, the value(s) of its operand(s), and the instruction ID; and
(3) *LLVM IR → instrumented library:* This step compiles the instrumented LLVM IR to an instrumented library. Any client program, such as our search module, can retrieve the runtime information of `sample.c` by invoking its instrumented version from step (2).

Since Atomu is dynamic and needs to instrument only the floating-point operations and focuses on critical atomic conditions, all remaining program constructs — such as loops, conditionals, casting, and I/O — do not need any specific treatments in our implementation, which is a distinct novelty and strength of our approach.

We have implemented in C++ the evolution algorithm (EA) as described in Section 4.3. The random variables used in the EA module are generated by `uniform_real_distribution` and `geometric_distribution` from C++'s random number generation facilities (*i.e.*, the `<random>` library). We set the initialization size to 100,000 and the iteration size to 10,000 on each potentially unstable operation. The σ_{st} and σ_{end} in the mutation step are set to 10^{-2} and 10^{-13}, respectively. These parameters are easily configurable.

5.2 Experimental Setup

5.2.1 Subjects. We conduct a set of experiments to evaluate Atomu on subjects chosen from the GNU Scientific Library (GSL),[4] version 2.5. GSL is an open-source numerical library that provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. GSL has been frequently used as test subjects in previous research [Barr et al.

[3] https://clang.llvm.org/
[4] https://www.gnu.org/software/gsl/

Table 3. Size of GSL functions and Atomu results.

Size on GSL Functions	FP Operations	Potentially Unstable Operations	Unstable Operations	Atomu Results
Average on 107 functions	87.8	38.7	11.1	11.1
Average on 88 functions	90.4	39.8	11.8	11.8
Total on 107 functions	9392	4141	1192	1192
Total on 88 functions	7957	3948	1037	1037

2013; Yi et al. 2019; Zou et al. 2015]. This version of GSL contains 154 functions with all floating-point parameters and return values. 107 (69%) of these 154 functions are univariate functions, which we choose as our experimental subjects. All the parameters and the return values of these 107 functions are of double precision (64 bits). Note that we conducted our experiments on univariate functions following existing work [Yi et al. 2019] for direct comparisons. Our approach is not limited to univariate functions and can be easily applied to multivariate functions by changing the search module's parameters.

5.2.2 Oracles. Although Atomu does not require oracles f (or high-precision results $\hat{f}_{high}$) during its complete process, we still need to know the accurate output values to validate the effectiveness of Atomu. To this end, we utilize *mpmath* [Johansson et al. 2018] to compute the oracles as the *mpmath* library supports floating-point arithmetic with arbitrary precision. Note that *mpmath* only supports a portion of the 107 functions and does so in two ways:

- *Direct support:* For example, "gsl_sf_bessel_I0(x)" corresponds to "besseli($0, x$)" in *mpmath*.
- *Indirect support via reimplementation:* For GSL functions without direct support, we reimplement them based on *mpmath* following their definitions from the GSL documentation. For example, "gsl_sf_bessel_K0_scaled(x)" in GSL is implemented by "besselk($0, x$) $\times$ exp(x)" in *mpmath*, where exp(x) is the scale term.

In total, 88 of the 107 functions are supported by *mpmath*, which we use as oracles.

5.2.3 Error Measurements and Significant Error. As discussed in Section 2.2, our evaluation uses relative error Err_{rel} as the error measurement, which is the prevalent measurement for floating-point errors [Goldberg 1991; Higham 2002]. Following existing work [Zou et al. 2015], we define a relative error greater than 0.1% ($Err_{rel} > 10^{-3}$) as significant.

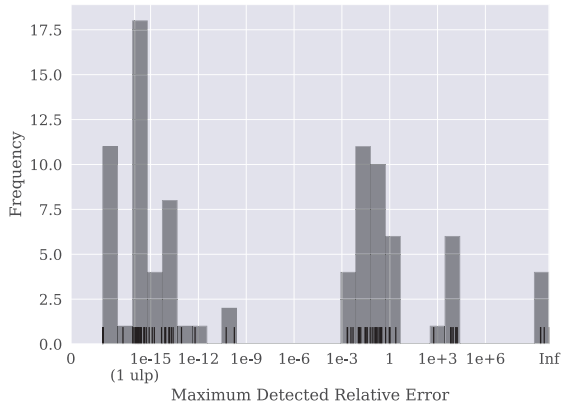
All our experiments are conducted on a desktop running Ubuntu 18.04 LTS with an Intel Core i7-8700 @ 3.20 GHz CPU and 16GB RAM.

5.3 Evaluation Results

This section presents evaluation results to show our approach's strong effectiveness and scalability over the state-of-the-art. In particular, we address the following research questions (RQs):

- **RQ1:** How effective is Atomu in detecting unstable operations?
- **RQ2:** How effective is Atomu in detecting functions with significant errors?
- **RQ3:** How scalable is Atomu?

5.3.1 RQ1: How Effective is Atomu in Detecting Unstable Operations? Table 3 shows the average size of GSL functions and the average size of results detected by Atomu, both in terms of the number of floating-point (FP) operations. The *FP Operations* column shows the average number of floating-point operations in the studied GSL functions. The *Potentially Unstable Operations* column



Maximum Detected Error	Number of Functions
$Err_{rel} \in [0, 10^{-15})$	31
$Err_{rel} \in [10^{-15}, 10^{-12})$	13
$Err_{rel} \in [10^{-12}, 10^{-9})$	2
$Err_{rel} \in [10^{-9}, 10^{-6})$	0
$Err_{rel} \in [10^{-6}, 10^{-3})$	0
Insignificant Error	46
$Err_{rel} \in [10^{-3}, 1)$	29
$Err_{rel} \in [1, 10^{3})$	3
$Err_{rel} \in [10^{3}, +\infty)$	10
Significant Error	42
Total	88

Fig. 3. Distribution of the largest detected relative errors on the 88 GSL functions.

shows that about 44% ($38.7/87.8 \approx 39.8/90.4 \approx 0.44$) of FP operations are potentially unstable (*e.g.*, $+, -$, sin, log, ..., as we defined in Section 4.2). The *Unstable Operations* column shows the number of FP operations that are found to be unstable, *i.e.*, triggering an atomic condition greater than 10 in at least one execution during the whole search process. Since for each unstable operation, Atomu keeps one input that triggers the largest atomic condition on it, the size of Atomu results is always the same as the number of unstable operations. Our data show that 13% ($11.1/87.8 \approx 11.8/90.4 \approx 0.13$) of the FP operations are indeed unstable.

5.3.2 RQ2: How Effective is Atomu *in Detecting Functions with Significant Errors?* Although Atomu can be run on all the 107 subject functions, since the oracles from *mpmath* are only available on 88 functions, we answer this RQ with results for the 88 functions.

Figure 3 shows the maximum detected errors on the 88 GSL functions. We observe that there are two peaks in this histogram, and it is easy to distinguish functions from these two peaks because there is a large gap from 10^{-9} to 10^{-3} with no function in it. The peak to the right consists of functions with $Err_{rel} > 10^{-3}$, thus, functions with significant errors, while the peak to the left consists of functions with small errors.

Figure 3 shows the detailed number of this distribution. We follow the definition in Section 5.2.3 that $Err_{rel} > 10^{-3}$ is defined as a significant error. We find that:

> Atomu detects that 42 (47.7%) of the 88 GSL functions have significant errors.

The state-of-the-art technique DEMC [Yi et al. 2019] uses high-precision results $\hat{f}_{high}$ to guide its search. The dataset for DEMC contains 49 GSL functions that are a subset of the 88 functions that we use as subjects. DEMC detects 20 among the 49 functions to have significant errors. Thus, we can directly compare Atomu and DEMC on these 49 functions. Since the reported data by DEMC uses *ErrBits* but not relative error (Err_{rel}), we re-calculate the *ErrBits* on our results.

Our results show that on the 49 functions dataset, Atomu can detect 28 functions to have significant errors, while DEMC only detects 20, a subset of the 28 detected by Atomu. Thus, the results show that Atomu is significantly more effective than DEMC in detecting significant errors. More details of the results are shown in Table 4 and Table 5.

To control the variance in running time, we repeated the same set of experiments 10 times and computed the standard deviation of Atomu's running time on each function. The average standard

Table 4. Data on the 42 functions with significant errors.

GSL Function Name	Error Triggering Input	Detected Relative Error	Time on Atomu (seconds)	Significant Error Detected by DEMC?	Time on DEMC (seconds)
gsl_sf_airy_Ai	-4.042852549222488e+11	6.64E+03	0.94	✓	112.35
gsl_sf_airy_Bi	-7.237129918123468e+11	2.89E+09	0.88	✓	91.37
gsl_sf_airy_Ai_scaled	-3.073966210399579e+11	1.40E+04	0.95	-	
gsl_sf_airy_Bi_scaled	-8.002750158072251e+11	4.91E+09	0.93	-	
gsl_sf_airy_Ai_deriv	-1.018792971647468e+00	3.70E-03	0.37	✓	203.47
gsl_sf_airy_Bi_deriv	-2.294439682614124e+00	2.20E-01	0.37	✓	188.82
gsl_sf_airy_Ai_deriv_scaled	-1.018792971647467e+00	1.09E-02	0.36	-	
gsl_sf_airy_Bi_deriv_scaled	-2.294439682614120e+00	1.26E-02	0.39	-	
gsl_sf_bessel_J0	2.404825557695774e+00	5.97E-02	0.68	✓	2079.15
gsl_sf_bessel_J1	3.831705970207514e+00	1.79E-01	0.72	✓	1777.88
gsl_sf_bessel_Y0	3.957678419314854e+00	7.93E-02	0.64	✓	325.22
gsl_sf_bessel_Y1	2.197141326031017e+00	1.04E-01	0.69	✓	753.16
gsl_sf_bessel_j1	-7.725251836937709e+00	2.17E-03	0.07	-	
gsl_sf_bessel_j2	9.095011330476359e+00	4.99E-03	0.07	-	
gsl_sf_bessel_y0	2.585919463588284e+17	1.72E+04	0.22	-	
gsl_sf_bessel_y1	9.361876298934626e+16	9.58E+03	0.56	-	
gsl_sf_bessel_y2	1.586407411088372e+17	1.46E+04	0.60	-	
gsl_sf_clausen	1.252935780352301e+14	9.36E-01	0.25	✓	471.55
gsl_sf_dilog	1.259517036984501e+01	5.52E-01	0.27	✗	459.64
gsl_sf_expint_E1	-3.725074107813663e-01	2.92E-02	0.43	✗	96.18
gsl_sf_expint_E2	-1.347155251069168e+00	2.40E+00	0.49	✗	165.38
gsl_sf_expint_E1_scaled	-3.725074107813663e-01	2.92E-02	0.63	-	
gsl_sf_expint_E2_scaled	-2.709975303391678e+228	3.01E+212	0.62	-	
gsl_sf_expint_Ei	3.725074107813668e-01	1.11E-02	0.44	✓	112.66
gsl_sf_expint_Ei_scaled	3.725074107813666e-01	1.41E-01	0.63	-	
gsl_sf_Chi	5.238225713898647e-01	1.28E-01	0.80	✓	199.98
gsl_sf_Ci	2.311778262696607e+17	5.74E+02	1.46	✓	84.80
gsl_sf_lngamma	-2.457024738220797e+00	3.06E-01	0.32	✗	106.87
gsl_sf_lambert_W0	1.666385643189201e-41	3.11E-01	0.11	✗	309.05
gsl_sf_lambert_Wm1	1.287978304826439e-121	1.00E+00	0.12	-	
gsl_sf_legendre_P2	-5.773502691896254e-01	3.81E-02	0.02	✓	1168.49
gsl_sf_legendre_P3	7.745966692414830e-01	3.72E-02	0.02	✓	908.69
gsl_sf_legendre_Q1	8.335565596009644e-01	1.28E-02	0.04	✓	995.65
gsl_sf_psi	-6.678418213073426e+00	9.89E-01	0.60	✓	187.66
gsl_sf_psi_1	-4.799999999999998e+01	1.40E-01	0.33	✓	165.71
gsl_sf_sin	-5.037566598712291e+17	2.90E+09	0.26	✗	135.14
gsl_sf_cos	-1.511080519199221e+17	7.96E+03	0.26	✗	130.22
gsl_sf_sinc	3.050995817918706e+15	1.00E+00	0.37	✗	149.43
gsl_sf_lnsinh	8.813735870195427e-01	2.64E-01	0.03	✓	236.93
gsl_sf_zeta	-9.999999999999984e+00	1.29E-02	0.98	✓	584.12
gsl_sf_zetam1	-1.699999999999999e+02	2.26E-03	1.11	-	
gsl_sf_eta	-9.999999999999989e+00	1.53E-02	1.05	✓	668.39
Average Time on Functions with Significant Errors			0.5		459.6
Average Time on All Supported Functions			0.34		585.8

derivation is 0.0060, the maximum one is 0.042, and the minimum one is 0.0007, indicating that Atomu's running time on each function is quite stable and does not vary significantly.

We also notice that the average *ErrBits* on significant error is 54.2 for Atomu and 57.5 for DEMC. The reason is that DEMC is guided by *ErrBits*, while Atomu searches for significant relative error

Table 5. Data on the 46 functions without significant errors.

GSL Function Name	Detected Relative Error	Time on Atomu (seconds)	Significant Error Detected by DEMC?	Time on DEMC (seconds)
gsl_sf_bessel_I0	2.37E-16	0.16	✗	191.23
gsl_sf_bessel_I1	2.21E-16	0.17	✗	189.81
gsl_sf_bessel_I0_scaled	1.92E-16	0.29	-	
gsl_sf_bessel_I1_scaled	0.00E+00	0.29	-	
gsl_sf_bessel_K0	2.64E-16	0.19	✗	3336.70
gsl_sf_bessel_K1	1.70E-16	0.20	✗	7905.00
gsl_sf_bessel_K0_scaled	1.44E-16	0.18	-	
gsl_sf_bessel_K1_scaled	1.69E-16	0.19	-	
gsl_sf_bessel_j0	1.12E-16	0.04	-	
gsl_sf_bessel_i0_scaled	0.00E+00	0.02	-	
gsl_sf_bessel_i1_scaled	4.87E-15	0.03	-	
gsl_sf_bessel_i2_scaled	0.00E+00	0.03	-	
gsl_sf_bessel_k0_scaled	0.00E+00	0.01	-	
gsl_sf_bessel_k1_scaled	0.00E+00	0.01	-	
gsl_sf_bessel_k2_scaled	0.00E+00	0.01	-	
gsl_sf_ellint_Kcomp	1.79E-10	0.37	✗	48.44
gsl_sf_ellint_Ecomp	1.27E-15	0.81	-	
gsl_sf_erfc	8.13E-16	0.28	✗	205.25
gsl_sf_log_erfc	5.37E-16	0.20	✗	295.88
gsl_sf_erf	9.10E-17	0.27	✗	71.04
gsl_sf_erf_Z	0.00E+00	0.02	-	
gsl_sf_erf_Q	8.64E-15	0.29	-	
gsl_sf_hazard	7.78E-15	0.23	-	
gsl_sf_exp	0.00E+00	0.01	✗	46.14
gsl_sf_expm1	2.58E-14	0.02	✗	39.11
gsl_sf_exprel	1.52E-14	0.02	-	
gsl_sf_exprel_2	5.51E-11	0.03	-	
gsl_sf_Shi	4.21E-16	0.63	✗	151.78
gsl_sf_Si	1.88E-16	0.56	✗	657.17
gsl_sf_fermi_dirac_m1	0.00E+00	0.02	-	
gsl_sf_fermi_dirac_0	1.51E-14	0.02	-	
gsl_sf_fermi_dirac_1	3.97E-16	0.30	-	
gsl_sf_fermi_dirac_2	3.82E-16	0.30	-	
gsl_sf_fermi_dirac_mhalf	1.72E-15	0.42	-	
gsl_sf_fermi_dirac_half	1.42E-14	0.44	-	
gsl_sf_fermi_dirac_3half	8.74E-15	0.41	-	
gsl_sf_gamma	1.99E-14	0.24	✗	197.16
gsl_sf_gammainv	8.67E-14	0.51	✗	207.41
gsl_sf_legendre_P1	0.00E+00	0.01	✗	416.00
gsl_sf_legendre_Q0	7.66E-17	0.04	✗	659.26
gsl_sf_log	1.11E-16	0.02	✗	154.74
gsl_sf_log_abs	1.85E-17	0.02	✗	245.43
gsl_sf_log_1plusx	1.30E-16	0.12	✗	120.50
gsl_sf_log_1plusx_mx	1.53E-16	0.11	✗	347.54
gsl_sf_synchrotron_2	6.16E-13	0.17	-	
gsl_sf_lncosh	0.00E+00	0.04	✗	441.09
Average Time on Functions without Significant Errors		0.19		758.4
Average Time on All Supported Functions		0.34		585.8

Err_{rel} and its results are re-calculated to *ErrBits* for comparison. Although both *ErrBits* and Err_{rel} may be used to measure floating-point errors, the relationship between these two measurements is not always consistent. Due to the lack of DEMC's raw data, we could only perform this comparison based on the reported *ErrBits* data from the published work on DEMC.

Section 5.3.2 shows the effectiveness of Atomu on detecting significant errors. It reports 42 functions with significant errors. This result is based on inspecting all inputs generated by Atomu, *i.e.*, inspecting on average 11.8 inputs for each function (see Table 3).

We have performed two additional analyses to show empirically that Atomu does not incur false positives nor false negatives on our evaluation subjects. First, we consider false positives. For example, the function "gsl_sf_expint_E2_scaled" is defined as $f(x) = e^x E_2(x)$, where $E_2(x)$ is the second-order exponential integral. Atomu reports a significant error on the function:

- Input: -2.709975303391678e+228
- Output result from GSL: 1.1102230246251565e-16
- Oracle result from *mpmath*: -3.690070528496872e-229
- Err_{rel}: 3.0086769779909848e+212

To validate that the oracle result from *mpmath* is accurate, we manually analyzed the series expansion of the function at $x = +\infty$:

$$f(x) = \frac{1}{x} - \frac{2}{x^2} + \frac{6}{x^3} + O\left(\left(\frac{1}{x}\right)^5\right)$$

We notice that $1/(-2.7099753 \times 10^{228}) \approx -3.69007 \times 10^{-229}$, which confirms that *mpmath* is accurate on the input and Atomu does find an error-triggering input -2.709975303391678e+228 on "gsl_sf_expint_E2_scaled."

In a further analysis, we choose the eight functions in Table 4 where Atomu detects significant errors while DEMC does not, and five additional functions at random from Table 4. Our analysis confirms empirically that Atomu does not incur false positives.

As for possible false negatives, we perform a related analysis to improve our confidence empirically that the functions in Table 5 do not have significant errors. We choose five functions at random and perform intensive testing. In particular, we run Atomu on each the five functions with both 1000x of the original initialization and iteration sizes. Our results show that the largest detected relative errors remain at the same magnitude across all five functions, providing strong evidence that Atomu, empirically, does not have false negatives.

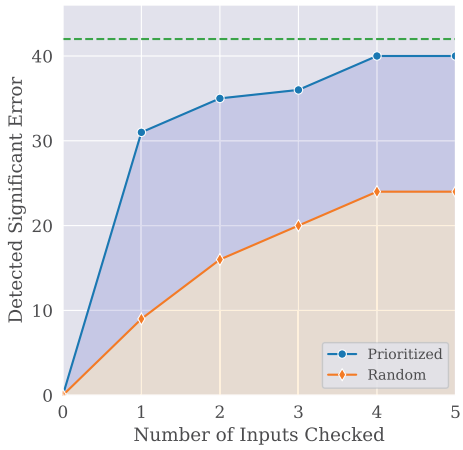


Fig. 4. Input ranking on functions with significant errors.

To further strengthen the usability of Atomu, we have proposed a method to rank inputs (see Section 4.4) with the goal of detecting significant errors by inspecting as few inputs as possible.

Figure 4 shows the effectiveness of our input ranking. The chart contains two lines, one for inspecting the inputs following the rank order, and the other for inspecting inputs randomly. The line chart shows that, by inspecting the top-1

ranked input, we can detect significant errors in 74% (31 over 42) of all functions with significant errors. This number raises to 95% by inspecting the top-4 ranked inputs. In contrast, as the baseline, randomly inspecting 1 and 4 inputs per function can detect 21% and 57% functions with significant errors, respectively.

> The input ranking method is effective, detecting 74% of the functions with significant errors using the top-1 ranked inputs and 95% using top-4 ranked inputs.

> Compared with the state-of-the-art technique, Atomu achieves a 40% improvement (28 vs. 20) in detecting significant errors.

5.3.3 RQ3: How Scalable is Atomu*?* Since Atomu does not rely on any high-precision computation $\hat{f}_{high}$, it is expected to be significantly faster than state-of-the-art techniques, which all require $\hat{f}_{high}$. This section presents strong empirical results to confirm our hypothesis.

For this comparison, we use the state-of-the-art DEMC [Yi et al. 2019], whose search method consists of a partitioned global search and a fine-grained search guided by the high-precision $\hat{f}_{high}$. We also compare with LSGA [Zou et al. 2015], another approach for detecting floating-point errors. It is meta-heuristic search-based using genetic algorithms, and also guided by $\hat{f}_{high}$ results.

Since the test subjects of DEMC and LSGA are also functions from GSL, we can compare the three approaches in terms of average time consumption. Atomu spends on average 0.34 seconds to analyze one GSL function, and the time to run the oracle on all inputs reported by Atomu for each GSL function is 0.09 seconds on average. Considering both the Atomu time and the oracle time, our approach is 1,362x faster than DEMC (585.8 seconds per function) and 140x faster than LSGA (~60 seconds per function). Note that DEMC depends on extra domain information while Atomu and LSGA do not. For example, on the function "gsl_sf_eta", DEMC only searches the domain [-168, 100], while Atomu and LSGA search the whole space of floating-point numbers, which is $(-1.8 \times 10^{308}, 1.8 \times 10^{308})$ for double precision. Thus, the comparison significantly favors DEMC, and even so, Atomu is still much faster than DEMC.

> Compared to the two state-of-the-art techniques, Atomu is 1,362x faster than DEMC and 140x faster than LSGA.

5.4 Case Study

This section details a case study on one of the 88 GSL functions: "gsl_sf_lngamma(x)". According to its documentation[5], it computes the logarithm of the Gamma function, $\log(\Gamma(x))$, subject to x not being a negative integer or zero. For $x < 0$, the real part of $\log(\Gamma(x))$ is returned, which is equivalent to $\log(|\Gamma(x)|)$. The function is computed using the real Lanczos method [Lanczos 1964].

Atomu reports that the input -2.457024738220797 triggers a significant relative error of 30.6%. To understand the root cause of this significant error, we manually analyzed the source code of this function. Figure 5 shows the simplified code in "gsl_sf_lngamma(x)" used to compute $x = -2.457024738220797$.

First, we explain the logic of this code snippet. The Lanczos method [Lanczos 1964] is an iteration method that can compute the Gamma function and the logarithm of Gamma function for $x > 0$. To support negative x, the GSL developers applied Euler's reflection formula [Silverman et al. 1972]

[5] https://www.gnu.org/software/gsl/doc/html/specfunc.html#gamma-functions

```
1  // inaccurate at x = -2.457024738220797
2  double gsl_sf_lngamma(const double x) {
3    // x < 0 while x is not near an integer.
4    if ( ... ) {
5      double z = 1.0 - x;
6      double lngamma_z = lngamma_lanczos(z);
7      double val = LOG_PI
8        - (log(fabs(sin(PI*z)))+lngamma_z);
9      return val;
10   }
11   else { ... } }
```

$$\Gamma(z)\Gamma(1-z) = \frac{\pi}{\sin(\pi z)}, z \notin \mathbb{Z} \tag{4}$$

$$\begin{aligned} &\log\left(\left|\Gamma(x)\right|\right) \\ &= \log\left(\left|\Gamma(1-z)\right|\right), x = 1-z, x < 0, x \notin \mathbb{Z} \\ &= \log\left(\left|\frac{\pi}{\Gamma(z)\cdot\sin(\pi z)}\right|\right) \\ &= \log(\pi) - \log\left(\left|\Gamma(z)\cdot\sin(\pi z)\right|\right) \\ &= \log(\pi) - \left(\log(|\Gamma(z)|) + \log(|\sin(\pi z)|)\right) \end{aligned} \tag{5}$$

Fig. 5. Simplified code in `gsl_sf_lngamma(x)`.

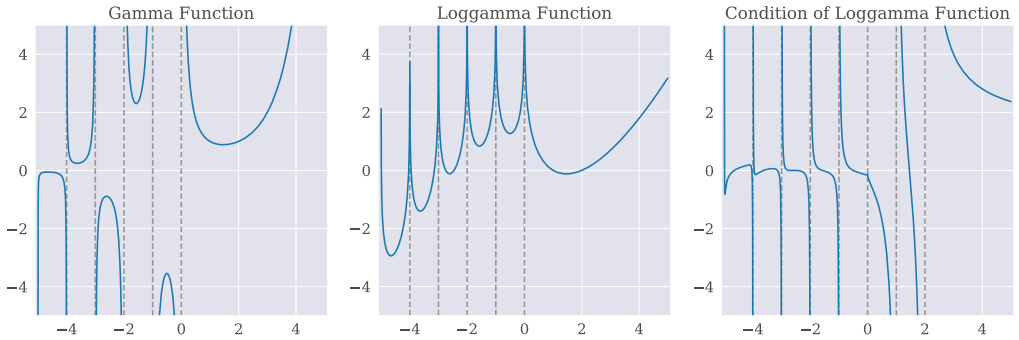


Fig. 6. Gamma, Loggamma and the global condition of Loggamma.

(in Equation (4)) to compute the Gamma of z, $z = 1 - x$ instead. Equation (5) shows the detailed inference of the formula used in Figure 5. We can see that line 6 uses the Lanczos method to compute the $\log(|\Gamma(z)|)$, and lines 7-8 compute the $\log(|\Gamma(x)|)$ exactly following the Equation (5).

Second, we explain how Atomu finds this error-triggering input −2.457024738220797.

(1) `LOG_PI` = 1.1447298858494002 is a hard-coded constant.
(2) `lngamma_z` = 1.153871662795107**8** contains a relative error about 1.58e-15.
(3) `log(fabs(sin(PI*z)))` = -0.0091417769457117**24** contains a relative error about 4.66e-15.
(4) `tmp = lngamma_z + log(fabs(sin(PI*z)))` = 1.144729885849**364** contains a relative error about 1.50e-15.
(5) `val = LOG_PI - tmp` = **3.774758283725532**e-15 contains a relative error of 3.06e-01. For this subtraction operation, its atomic condition is 6.06e+14. The small error in `tmp` is significantly amplified by this critical atomic condition.

When Atomu searches on the last subtraction operation, it tries to generate inputs triggering critical atomic conditions, and finally finds $x = -2.457024738220797$ that triggers the largest atomic condition on this operation.

Third, we notice that "gsl_sf_lngamma(x)" is included in the benchmarks of DEMC, but DEMC did not detect significant errors on this function. DEMC applies its estimated global condition to guide its search. However, Figure 6 shows that the global condition of Loggamma function around $x = -2.457024738220797$ is near 0. For this reason, DEMC will not search around this domain and cannot detect the significant error. This case also shows that using atomic conditions can be more powerful than global conditions.

This case study shows that even expert-written libraries, with carefully fine-tuned algorithms (both the Lanczos method and Euler's reflection), still suffer from inaccuracy problems.

5.5 Discussions

This section provides further details and discussions on the technical design and development of our approach. In particular, we elaborate on the challenges on using the high-precision program $\hat{f}_{high}$, the benefits of using atomic vs. global conditions, the reasons of using relative error for measurement, and risks of estimating global conditions without high-precision implementations.

5.5.1 Challenges for Using High-Precision Floating-Point Implementations. In numerical analysis, it is common to use a high-precision program $\hat{f}_{high}$ to simulate the conceptional mathematical function f [Bao and Zhang 2013; Benz et al. 2012; Fu et al. 2015]. However, as discussed in Section 1, it is costly to use $\hat{f}_{high}$ in terms of both runtime and development cost, which we further elaborate on.

In terms of runtime cost, even quadruple precision (128 bits) is 100x slower than double precision (64 bits) [Peter Larsson 2013], while programs using arbitrary precision libraries, such as MPFR [Fousse et al. 2007] and mpmath [Johansson et al. 2018], incur further slowdowns when they use increased precision.

In terms of development cost, high-precision implementations need to deal with precision-specific operations [Wang et al. 2016] and hard-coded series expansions (*cf.* examples in Section 1). Such patterns occur frequently in numerical library functions, such as "gsl_sf_bessel_j0", "gsl_sf_log_erfc", "gsl_sf_log_1plusx", *etc.*. Another example is using hard-coded iterations. For example, a program uses Newton's method to find roots of $f(x)$ as follows:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The program loops for a hard-coded number of iterations or uses a hard-coded tolerance. To make this kind of programs more accurate, expert knowledge is needed to manually tune such parameters. Without carefully tackling these challenges, $\hat{f}_{high}$ cannot be treated as a high-quality approximation to f. Due to the lack of automated tools to handle these problems, implementing a high-precision program $\hat{f}_{high}$ incurs high development cost, if not practically infeasible.

These aforementioned challenges motivate our approach of atomic conditions.

5.5.2 Atomic Condition vs. Global Condition. Several techniques [Fu et al. 2015; Yi et al. 2017] propose the use of global conditions to analyze the (in)accuracy of floating-point programs. They treat the given program as a black-box and compute its global conditions to measure the program's (in)stability [Fu et al. 2015].

Compared with global conditions, atomic conditions bring several important benefits which are summarized in Table 6 and further discussed below:

- **Speed**: Atomic conditions can be straightforwardly computed by pre-calculated formulae. Computing global conditions, on the other hand, involves high-precision $\hat{f}_{high}$ [Fu et al. 2015], which introduces both high runtime overhead and expensive development cost.
- **Soundness**: By using the pre-calculated formulae, atomic conditions are guaranteed to be unbiased. Previous work [Yi et al. 2017] has suggested to estimating global conditions without using high-precision implementations. However, this estimation can be biased and infeasible (bias becomes the dominant term) as we will discuss in more detail in Section 5.5.4.

Table 6. Benefits from atomic conditions.

	Atomic Condition	Global Condition (Accurate)	Global Condition (Estimated)
Speed	✓	✗	✓
Soundness	✓	✓	✗
Interpretability	✓	✗	✗

- **Interpretability**: Atomic conditions help explain how errors are introduced and amplified by atomic operations. If a significant error occurs in a program's result, one can easily locate the responsible atomic operation with a significantly large atomic condition.

We also use an intuitive example to illustrate the advantages of atomic conditions. Recall the motivating example in Section 3, where we have a program $\hat{f}(x)$ that triggers significant errors when x is a small number close to 0:

$$f(x) = \frac{1-\cos(x)}{x^2} \qquad \lim_{x\to 0} f(x) = \frac{1}{2}$$

Note that when the input $x = 10^{-7}$, the atomic condition on the subtraction becomes significant, thus the subtraction is to blame for the significant error in the computation result.

However, the global condition is 0 when x approaches 0:

$$\Gamma_f(x) = \frac{x\sin(x) + 2\cos(x) - 2}{1-\cos(x)} \qquad \lim_{x\to 0} \Gamma_f(x) = 0$$

This suggests that approaches based on global conditions to detect inaccuracies may be ineffective on this program.

5.5.3 The Reason for Using Relative Error for Measurement. As mentioned in Section 2.2, relative error is the prevalent measurement for floating-point errors. There is also the measurement of *ErrBits*, which we have mentioned in Section 5.3 when comparing our approach with DEMC [Yi et al. 2019]. *ErrCount* is defined as the count of floating-point numbers ($\mathbb{F}$) between the ideal $f(x)$ and the floating-point result $\hat{f}(x)$, and *ErrBits* is defined as the logarithm of *ErrCount* to base 2:

$$ErrCount(f(x), \hat{f}(x)) = \left|\left\{p \in \mathbb{F} \middle| \min(f(\mathbf{x}), \hat{f}(\mathbf{x})) < p \le \max(f(\mathbf{x}), \hat{f}(\mathbf{x}))\right\}\right|$$
$$ErrBits(f(x), \hat{f}(x)) = \log_2\left(ErrCount\big(f(x), \hat{f}(x)\big)\right)$$

There are two clear drawbacks to this measurement. First, it is quite counter-intuitive. In other words, the measurement is inconsistent over the whole floating-point domain due to the distribution of floating-point numbers. For example, they are similar for two very different intervals: $ErrBits(0,1) = 62$ and $ErrBits(0, 10^{-152}) = 61$, while quite different for two similar intervals: $ErrBits(0,1) = 62$ and $ErrBits(1,2) = 52$.

Second, it does not distinguish between oracle and program results, while relative errors do. For example, we have $ErrBits(f = 0.01, \hat{f} = 1) = 54.75$, while also $ErrBits(f = 1, \hat{f} = 0.01) = 54.75$. On the other hand, we have $Err_{rel}(f = 0.01, \hat{f} = 1) = 99$, while $Err_{rel}(f = 1, \hat{f} = 0.01) = 0.99$, so relative error can distinguish these two scenarios.

5.5.4 Estimating Global Condition without High-Precision Implementations. For a numerical program $\hat{f}$ under analysis, in most cases, the derivative $f'(x)$ is unavailable as a mathematical expression. Thus, the derivative $f'(x)$ used in computing the global condition $\Gamma_f(x) = \left|\frac{x\cdot f'(x)}{f(x)}\right|$ can only

be estimated by

$$f'(x) = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta} \tag{6}$$

Because of the absence of a high-precision implementation, there exist relative errors ε between the mathematical function f and the numerical program $\hat{f}$:

$$\begin{aligned} \hat{f}(x) &= f(x) \pm \varepsilon_1 f(x) \\ \hat{f}(x+\delta) &= f(x+\delta) \pm \varepsilon_2 f(x+\delta) \end{aligned} \tag{7}$$

The estimation of derivative $\hat{f}'(x)$ is computed as:

$$\begin{aligned} \hat{f}'(x) &= \frac{\hat{f}(x+\delta) - \hat{f}(x)}{\delta} \\ &= \frac{f(x+\delta) - f(x)}{\delta} \pm \frac{\varepsilon_1}{\delta} f(x) \pm \frac{\varepsilon_2}{\delta} f(x+\delta) \\ &\approx f'(x) \pm \frac{\varepsilon_1}{\delta} f(x) \pm \frac{\varepsilon_2}{\delta}(f(x) + \delta f'(x)), \text{ when } \delta \to 0. \\ &= f'(x) \pm \varepsilon_2 f'(x) \pm \frac{\varepsilon_1 \pm \varepsilon_2}{\delta} f(x) \end{aligned} \tag{8}$$

So, based on Equation (8), the estimation of the global condition $\hat{\Gamma_f}(x)$ is computed as:

$$\begin{aligned} \hat{\Gamma_f}(x) &= \left| \frac{x \cdot \hat{f}'(x)}{\hat{f}(x)} \right| \\ &= \left| \frac{(1 \pm \varepsilon_2)}{(1 \pm \varepsilon_1)} \cdot \frac{x \cdot f'(x)}{f(x)} \right| \pm \left| x \cdot \frac{\varepsilon_1 \pm \varepsilon_2}{\delta} \right| \\ &\approx \Gamma_f(x) \pm \left| x \cdot \frac{\varepsilon_1 \pm \varepsilon_2}{\delta} \right| \end{aligned} \tag{9}$$

To make Equations (6) and (8) hold, the δ should be small ($\delta \to 0$). At the same time, Equation (9) shows that the estimation is biased, and the bias term in the estimated global condition is given as:

$$\lim_{\delta \to 0} \left| x \cdot \frac{\varepsilon_1 \pm \varepsilon_2}{\delta} \right| = +\infty$$

which means without the high-precision program $\hat{f}_{high}$ to make the error $(\varepsilon_1 \pm \varepsilon_2) \ll \delta$, the bias term can become the dominant term in the estimation.

6 RELATED WORK

This section surveys several threads of closely-related work, which we discuss below.

Obtaining the oracle of floating-point programs. FpDebug [Benz et al. 2012] is built on MPFR [Fousse et al. 2007] and Valgrind [Nethercote and Seward 2007]. It dynamically analyzes a program by performing all floating-point instructions side-by-side in high precision. This type of approach assumes that the semantics of floating-point code in high precision is closer to that of the underlying mathematical function. However, this assumption does not hold on precision-specific operations [Wang et al. 2016] and precision-related code (Section 5.5.1). Thus, it remains a significant challenge to obtain oracles for floating-point programs.

Searching for error-triggering inputs. Several approaches have been proposed to searching inputs that trigger significant floating-point errors. BGRT [Chiang et al. 2014] is based on a heuristic binary search, LSGA [Zou et al. 2015] is based on a genetic algorithm, and DEMC [Yi et al. 2019] is

based on differential evolution and Markov Chain Monte Carlo (MCMC) methods. All such previous approaches rely on the existence of oracles.

Compared with these previous search-based techniques, our approach Atomu does not rely on the existence of oracles, which, as we have discussed, are expensive to obtain in general. Atomu reports a set of suspicious inputs. If an oracle exists, Atomu validates the suspicious inputs and reports those inputs that trigger significant errors. If an oracle does not exist, Atomu reports a ranked list of inputs, and we have shown empirically that 95% of the buggy functions can be found by inspecting the top-4 ranked inputs.

Localizing the root cause and repairing floating-point errors. Given a floating-point program and an error-triggering input, Herbgrind [Sanchez-Stern et al. 2018] can locate an expression, which is the root cause of error, for diagnosing and debugging. Given a small floating-point expression ($\approx$ 10 LoC), Herbie [Panchekha et al. 2015] can rewrite the expression to improve its numerical accuracy. However, since Herbie uses only 256 randomly sampled inputs, it may be unable to find the significant errors in the expression, thus would be unable to rewrite it. Herbgrind and Herbie can be combined and considered as an automated repair tool, but rely on a given error-triggering input. AutoRNP [Yi et al. 2019] proposes the DEMC and PTB algorithms to localize the error-triggering interval of inputs, and applies linear approximation to repair the localized interval.

Our approach Atomu can help localize the root cause of a significant error — it reports not only the error-triggering input, but also the operation on which a significant atomic condition has been triggered. This operation is the root cause of the significant error, *i.e.*, where the error is amplified significantly by its atomic condition. For program repair, the inputs reported by Atomu can be used by approaches that demand error-triggering inputs, such as the combined Herbgrind/Herbie.

Conditioning. Wilkinson introduced condition number for measuring the inherent stability of a mathematical function f [Higham 2002]. Recently, Fu et al. [2015] proposed an approach to computing the global condition and analyzing the accuracy of floating-point programs. Yi et al. [2019] proposed to use the estimated global conditions to measure floating-point errors.

Our work is the first white-box analysis that proposes an error propagation model based on atomic conditions. It is rooted on the insight that atomic conditions are the dominant factors of floating-point errors. Atomic conditions also have strong advantages in speed, soundness, and interpretability comparing with global conditions. Our empirical evaluation demonstrates that Atomu is highly effective and efficient, making accurate error analysis practically feasible.

Error-bound analysis. Several approaches have been proposed to *statically* analyze possible upper bounds on floating-point errors [Goubault and Putot 2011; Izycheva and Darulova 2017; Lee et al. 2016; Solovyev et al. 2019]. Such static error-bound analyses explicitly model floating-point errors as intervals [Hickey et al. 2001] and apply standard program analysis techniques, such as abstract interpretation or symbolic reasoning, to obtain possible upper error bounds on the program output.

Compared with these static error-bound analyses, our approach has several key differences: (1) Our approach is *dynamic*, while existing error-bound analyses are *static*; (2) our work introduces the concept of atomic condition; (3) our error propagation model is based on atomic conditions, and formulates the insight that atomic conditions are dominant factors for floating-point errors; and (4) our goal is different — while error-bound analysis focuses on estimating worst-case global error-bounds, we do not use the propagation model to estimate the global error, but focus on exploiting local/atomic conditions to help effectively find specific error-triggering inputs.

7 CONCLUSION

We have introduced a new, effective approach for finding significant floating-point errors in numerical programs. Our key insight is to rigorously analyze the condition numbers of the atomic

mathematical operations in numerical programs and use them to effectively guide the search for inputs that trigger large errors. We have designed and realized a general approach based on this insight, and extensively evaluated it on code from the widely-used GNU Scientific Library (GSL). Evaluation results have demonstrated the effectiveness of our approach — compared to state-of-the-art approaches, it not only precisely detects more GSL functions with large floating-point errors, but also does so several orders of magnitude faster, thus making error analysis significantly more practical. We expect the methodology and principles behind our approach to benefit other floating-point program analysis tasks such as debugging, repair and synthesis.

ACKNOWLEDGMENTS

We thank Pinjia He, Clara Meister, Manuel Rigger, Theodoros Theodoridis, Sverrir Thorgeirsson, Dominik Winterer, and the anonymous POPL reviewers for valuable feedback on earlier versions of this paper. This material is based upon work supported in part by the National Key Research and Development Program of China under Grant No. 2017YFB1001803, the National Natural Science Foundation of China under Grant No. 61922003, 61672045, the China Scholarships Council under Grant No. 201806010265, and the EU's H2020 Program under Grant No. 732287.

REFERENCES

Thomas Bäck, David B Fogel, and Zbigniew Michalewicz. 1999. *Evolutionary computation 1: Basic algorithms and operators* (1st ed.). CRC press.

James E. Baker. 1985. Adaptive Selection Methods for Genetic Algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms, Pittsburgh, PA, USA, July 1985*, John J. Grefenstette (Ed.). Lawrence Erlbaum Associates, 101–111.

Tao Bao and Xiangyu Zhang. 2013. On-the-fly detection of instability problems in floating-point program execution. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 817–832.

Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 549–560.

Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A dynamic program analysis to find floating-point accuracy problems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 453–462.

Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 43–52.

Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing probabilistic programming systems. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*. 574–586.

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 (2007), 13.

Zhoulai Fu, Zhaojun Bai, and Zhendong Su. 2015. Automated backward error analysis for numerical code. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 639–654.

David Goldberg. 1991. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Comput. Surv.* 23, 1 (1991), 5–48.

Eric Goubault and Sylvie Putot. 2011. Static Analysis of Finite Precision Computations. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings (Lecture Notes in Computer Science)*, Ranjit Jhala and David A. Schmidt (Eds.), Vol. 6538. Springer, 232–247.

John Harrison. 2009. Decimal Transcendentals via Binary. In *19th IEEE Symposium on Computer Arithmetic (ARITH)*, Javier D. Bruguera, Marius Cornea, Debjit Das Sarma, and John Harrison (Eds.). 187–194.

Timothy J. Hickey, Qun Ju, and Maarten H. van Emden. 2001. Interval arithmetic: From principles to implementation. *J. ACM* 48, 5 (2001), 1038–1068.

Nicholas J. Higham. 2002. *Accuracy and stability of numerical algorithms (2. ed.)*. SIAM.

Anastasiia Izycheva and Eva Darulova. 2017. On sound relative error bounds for floating-point arithmetic. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, Daryl Stewart and Georg Weissenbacher (Eds.). IEEE, 15–22.

Fredrik Johansson et al. 2018. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 1.1.0)*. `http://mpmath.org/`.

Morris Kline. 1998. *Calculus: an intuitive and physical approach*. Courier Corporation.

Cornelius Lanczos. 1964. A precision approximation of the gamma function. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis* 1, 1 (1964), 86–96.

Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. 75–88.

Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2016. Verifying bit-manipulations of floating-point. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 70–84.

Jacques-Louis Lions, Lennart Luebeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. 1996. Ariane 5 flight 501 failure report by the inquiry board.

Sandra Loosemore, Richard M Stallman, Rolandand McGrath, Andrew Oram, and Ulrich Drepper. 2019. The GNU C Library Reference Manual. (2019). https://www.gnu.org/software/libc/manual/html_node/Errors-in-Math-Functions.html

Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). 89–100.

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 (PLDI)*. 1–11.

Peter Larsson. 2013. Exploring Quadruple Precision Floating Point Numbers in GCC and ICC. https://www.nsc.liu.se/~pla/blog/2013/10/17/quadruple-precision/. [Online; accessed 24-June-2019].

Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *Proceedings of the 41st International Conference on Software Engineering, ICSE*. 1027–1038.

Kevin Quinn. 1983. Ever had problems rounding off figures. *This stock exchange has. The Wall Street Journal* (1983), 37.

Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding root causes of floating point error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 256–269.

Richard A Silverman et al. 1972. *Special functions and their applications*. Courier Corporation.

Robert Skeel. 1992. Roundoff error and the Patriot missile. *SIAM News* 25, 4 (1992), 11.

Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. 2019. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst.* 41, 1 (2019), 2:1–2:39.

Ran Wang, Daming Zou, Xinrui He, Yingfei Xiong, Lu Zhang, and Gang Huang. 2016. Detecting and fixing precision-specific operations for measuring floating-point errors. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 619–630.

Debora Weber-Wulff. 1992. Rounding error changes Parliament makeup. *The Risks Digest* 13, 37 (1992).

L. Darrell Whitley. 1989. The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best. In *Proceedings of the 3rd International Conference on Genetic Algorithms, George Mason University, Fairfax, Virginia, USA, June 1989*, J. David Schaffer (Ed.). Morgan Kaufmann, 116–123.

Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2017. Efficient Global Search for Inputs Triggering High Floating-Point Inaccuracies. In *24th Asia-Pacific Software Engineering Conference (APSEC)*. 11–20.

Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient Automated Repair of High Floating-point Errors in Numerical Libraries. *Proc. ACM Program. Lang.* 3, POPL, Article 56 (Jan. 2019), 29 pages.

Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. 2015. A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies. In *37th IEEE/ACM International Conference on Software Engineering (ICSE)*. 529–539.

Dan Zuras, Mike Cowlishaw, et al. 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70.