OFP-Repair: Repairing Floating-point Errors via Original-Precision Arithmetic

Youshuai Tan
The Hong Kong University of Science and Technology (Guangzhou)
China
ytan387@connect.hkust-gz.edu.cn

Zishuo Ding*
The Hong Kong University of Science and Technology (Guangzhou)
China
zishuoding@hkust-gz.edu.cn

Jinfu Chen
Wuhan University
China
jinfuchen@whu.edu.cn

Weiyi Shang
University of Waterloo
Canada
wshang@uwaterloo.ca

Abstract

Errors in floating-point programs can lead to severe consequences, particularly in critical domains such as military, aerospace, and financial systems, making their repair a crucial research problem. In practice, some errors can be fixed using original-precision arithmetic, while others require high-precision computation. Developers often avoid addressing the latter due to excessive computational resources required. However, they sometimes struggle to distinguish between these two types of errors, and existing repair tools fail to assist in this differentiation. Most current repair tools rely on high-precision implementations, which are time-consuming to develop and demand specialized expertise. Although a few tools do not require high-precision programs, they can only fix a limited subset of errors or produce suboptimal results.

To address these challenges, we propose a novel method, named OFP-Repair. Our method identifies original-precision-repairable errors by computing the condition number of the program with respect to its inputs. We then construct a unified repair framework using series expansion. ACESO is the only existing tool that neither requires nor relies on high-precision programs for both detection and repair, making it our primary baseline. On ACESO's dataset, our patches achieve improvements of three, seven, three, and eight orders of magnitude across four accuracy metrics. In real-world cases, our method successfully detects all five original-precision-repairable errors and fixes three, whereas ACESO only repairs one. Notably, these results are based on verified data and do not fully capture the potential of OFP-Repair. To further validate our method, we deploy it on a decade-old open bug report from GNU Scientific Library (GSL), successfully repairing five out of 15 bugs. The developers have expressed interest in our method and are considering integrating our tool into their development workflow. We are currently working on applying our patches to GSL. The results are highly encouraging, demonstrating the practical applicability of our technique.

*Corresponding author.

Conference'17, Washington, DC, USA
2025. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

CCS Concepts

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

Keywords

Floating-point Error, Program Repair, Numerical Bug, Real-world Project

1 Introduction

Floating-point programs are the basis of modern science and engineering and their errors can lead to serious consequences, including military applications [21], aerospace engineering [12], and financial systems [19, 28]. Therefore, many works have been conducted to detect the potential floating-point errors [1, 13, 30–32, 36]. Upon error detection, subsequent repair proves equally critical.

In practical scenarios, developers address numerical accuracy errors through two distinct strategies [5]: 1) utilizing high-precision floating-point arithmetic; and 2) restructuring numerical expressions while maintaining the original computational precision. Developers generally avoid increasing numerical precision as a corrective measure, as this approach necessitates substantial computational resources and significantly prolongs execution time [11]. For example, NumPy issue #1063 was closed without applying the fix because higher precision is too expensive [5]. In contrast, developers fix the SciPy issue #3545 by utilizing different but mathematically equal expressions. However, although developers prefer the cases that can be repaired using original-precision arithmetic, no currently available instrumentation facilitates developers' assessment of whether errors are repairable through original-precision numerical computations alone. For example, the developers initially attempted to resolve SciPy issue #6368 by modifying computational expressions, but subsequent results revealed that the bug could only be fixed with higher-precision arithmetic.

Although prior research has proposed many methods for repairing floating-point errors, they lack the capability to effectively identify and resolve the errors that can be repaired using only original-precision arithmetic: **1) Limitation 1: Both the detection and repair processes necessitate the use of high-precision programs.** Transforming original programs into high-precision

versions needs profound knowledge of math and numerical analysis [25]. **2) Limitation 2: Absence of a unified repair paradigm for original-precision-repairable errors.** Although some tools [4, 17, 26, 29] utilize mathematically identical transformations to repair errors, they are only capable of handling some of the original-precision-repairable errors because they rely heavily on templates. **3) Limitation 3: Lack of diagnostic capability for such errors.** This pain point manifests concretely in practical development scenarios, as demonstrated by SciPy issue #6368.

To address these limitations, we propose OFP-Repair (**Repair F**loating-**P**oint errors by using **O**riginal-precision arithmetic). We employ finite difference methods to compute numerical derivatives and derive the condition numbers of the target programs, thereby assessing whether the bug can be repaired by using original-precision computations. Subsequently, our method uses Taylor series to construct patches.

Since ACESO is currently the only approach that does not employ high-precision program repair, we evaluate our method's performance by comparing it with ACESO on the latter's dataset. Experimental results demonstrate that our method can successfully identify all the original-precision-repairable errors. Moreover, our patches achieve significantly higher precision than those of ACESO across all evaluated metrics. Specifically, the average maximum absolute error of all functions on the stable area is 4.11×10^{-16} (vs. ACESO's 2.45×10^{-13}), demonstrating a three-order-of-magnitude improvement. For maximum relative error on the stable area, we attain 7.47×10^{-16} compared to ACESO's 2.74×10^{-9}, representing a seven-fold enhancement in precision. Similarly, in the decayed area, our maximum absolute error of 2.13×10^{-16} outperforms ACESO's 2.45×10^{-13} by three orders of magnitude. Our maximum relative error of 3.73×10^{-15} in the decayed region shows an eight-order-of-magnitude improvement over ACESO's 5.74×10^{-07}. We further evaluate our method on five real-world original-precision-repairable errors collected by Franco et al. [5]. We can identify all five bugs and successfully repair three of them. In contrast, ACESO is only able to fix one bug.

In addition to applying our method to previously validated original-precision-repairable errors, we also attempt to repair open bug reports. We believe this evaluation strategy better demonstrates the effectiveness of our method, as successfully repairing such bugs suggests that our method is highly effective and has the potential for practical adoption in future software development. We use our method to repair bugs from an open GNU Scientific Library (GSL) bug report that was posted over a decade ago [1]. OFP-Repair is able to reduce the numerical errors in five out of the 15 bugs reported in this bug report. For three of these bugs, we successfully reduce the relative error to below 1×10^{-16}. For the other two bugs, we reduce the relative error by one and 15 orders of magnitude, respectively. These results indicate that, if our method had been available to developers at the time, the bugs could likely have been resolved instead of remaining open for more than ten years. The GSL developers have shown interest in adopting our method.

[1] http://savannah.gnu.org/bugs/?43259

The replication package, including the detection process and our patches, is available at the following link [2]. In summary, our work makes three main contributions:

- We introduce OFP-Repair, a novel method designed to effectively detect original-precision-repairable errors and repair them.
- We evaluate OFP-Repair on a diverse set of benchmarks, including datasets of ACESO and real-world bugs from Franco et al. [5].
- We successfully reduce relative errors of five bugs from a bug report that has remained open for over a decade.

To ensure clarity of key concepts utilized in this manuscript, we present the following definitions: In this context, we define the **original-precision-repairable errors** as the floating-point errors that can be fixed by using original-precision floating-point representations.

Work organization. The remainder of this work is structured as follows. Section 2 provides the necessary background for our research. In Section 3, we introduce the motivation of our work and discuss the shortcomings of current approaches. Section 4 details the design and implementation of OFP-Repair. The experimental evaluation is presented in Section 5. We present the results of our method when applied to the bugs in the bug report in Section 6. We analyze potential limitations in Section 7 and review related work in Section 8. Finally, Section 9 summarizes our contributions and concludes the work.

2 Background

In this section, we provide an overview of the background related to Taylor series, floating-point representation, and cancellation.

2.1 Taylor series

Taylor series are widely used in science and engineering [6, 23]. Equation 1 represents the Taylor series of the function f at a. The Taylor series can be employed to approximate functions in the vicinity of the point a based on the initial several terms, exhibiting a rapid rate of convergence [23]. Programmers prefer using polynomials to approximate the values of functions, as polynomials are among the simplest types of functions. For example, if we calculate the value of $\sin(x)$ for x near 0, we have: $\sin(x) = \sin(0) + \frac{\cos(0)}{1!}(x-0) + \frac{-\sin(0)}{2!}(x-0)^2 + \frac{-\cos(0)}{3!}(x-0)^3 + \frac{\sin(0)}{4!}(x-0)^4 + \frac{\cos(0)}{5!}(x-0)^5 \cdots$. Since we can directly obtain the values of $\sin(0)$ and $\cos(0)$, the equation becomes: $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots$. Figure 1 shows that retaining more terms in the Taylor series leads to better approximation performance.

$$
\begin{aligned}
f(x) &= \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \\
&= f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots
\end{aligned}
\tag{1}
$$

The use of Taylor series for function approximation is a common approach in numerical software. In the GNU Scientific Library [3], the

[2] https://github.com/abs-hello/fixing

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

Table 1: The bit allocation of IEEE standard floating-point numbers.

	Sign	Exponent	Significand
Single precision (32 bits)	1	8	23
Double precision (64 bits)	1	11	52

gsl_sf_sin_e function returns results computed using its Taylor series for inputs with an absolute value less than 1.2207031250000000e-04. Since only the first two terms of the Taylor series are retained, the applicable domain of the Taylor approximation in *gsl_sf_sin_e* is limited.

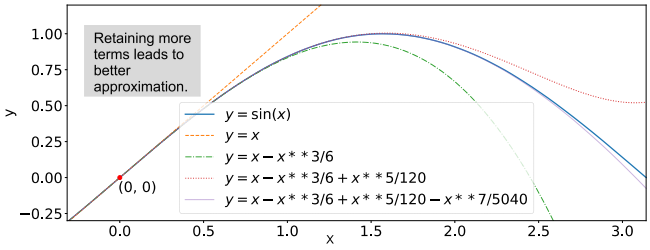

Figure 1: The approximation performance of Taylor series.

2.2 Floating-point representation

Floating-point arithmetic had become widely adopted as a standard computational method by the mid-1950s [16]. To achieve a unified floating-point system, a standard for binary floating-point representation and arithmetic was established through the cooperation between academic computer scientists and microprocessor chip designers. The standard was published in 1985, which was known as IEEE (Institute for Electrical and Electronics Engineers) 754.

The IEEE floating-point numbers can be represented in a standardized format consisting of three key components:

$$\pm(b_0.b_1b_2\ldots b_{p-1})_2 \times 2^E, \text{ where} \tag{2}$$

- The sign ± indicates whether the number is positive or negative;
- The binary fraction $b_0.b_1b_2\ldots b_{p-1}$ is the significand (or mantissa) part, with precision p. In the case of normalized numbers, the leading bit b_0 is implicitly 1;
- The exponent E is computed by $E = e - bias$, where e is the biased m-bit exponent value, and the $bias$ is given by $bias = 2^{m-1} - 1$.

The IEEE standard defines two basic representation formats, *single* and *double*. Table 1 illustrates the format of the three parts. The standard also advocates the inclusion of an extended precision format, characterized by a minimum of 15 bits allocated to the exponent and at least 63 bits dedicated to the significand.

2.3 Cancellation

Cancellation, which arises when performing subtraction between two nearly identical floating-point numbers [16], is one kind of operation which can result in significant errors for floating-point programs [36]. Analogously, it can also occur during the addition of two numbers with opposing signs but approximately equal magnitudes. For example, the difference between the two real numbers, $x = 3.14159265358973$ and $y = 3.14159265358972$, is 1×10^{-14}. However, if we conduct the subtraction operation using double-precision floating-point numbers, the result is $1.021405182655144 \times 10^{-14}$, leading to a relative error of about 0.0214, which is significant in many scientific applications. The observed accuracy degradation can be primarily ascribed to the constrained bit allocation for the significand component. As evidenced in Figure 2, a substantial cancellation of 44 significand bits occurs, resulting in merely eight effective precision bits remaining. In such a case, the computed result contains only eight correct bits in the significand component, whereas a double-precision floating-point number retains 52 significand bits. Consequently, the relative error between the computed result and the oracle value becomes substantial.

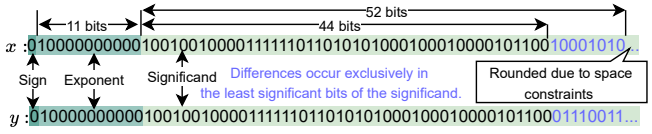

Figure 2: The double-precision floating-point representations of x and y.

3 Motivation

The first systematic study of numerical bugs in real-world software systems was carried out by Franco et al. [5]. This section first presents our research motivation emerging from this systematic study. Then we introduce an example to demonstrate that certain floating-point errors can be effectively repaired using original-precision floating-point numbers without resorting to high-precision floating-point numbers. Finally, we examine existing error-repairing methods, highlighting their limitations in repairing such original-precision-repairable errors.

Repairing floating-point errors without resorting to high-precision arithmetic represents a significant advancement, as high-precision programs developments typically involve substantial work [25] and high-precision programs tend to cost prolonged time [11]. Furthermore, certain computational environments may not support high-precision floating-point representations.

3.1 Why choose our method? Insights from a real-world numerical bugs study

Franco et al. [5] have conducted the first systematic investigation of numerical bugs encountered in real-world software systems, including NumPy[4], SciPy[5], LAPACK[6], GNU Scientific Library[7], and Elemental [18]. Through rigorous analysis, they identified 269 numerical bugs and developed a novel taxonomy for classifying these bugs. One significant category is accuracy bugs due to rounding or truncation errors. Their detrimental effects are substantial, as they evade detection by compilers (e.g., underflow conditions), potentially leading users to obtain erroneous results. Based on their exploratory findings, developers address the accuracy errors through two primary approaches: one involves repairing the bug while maintaining the original precision; the other requires increasing the floating-point precision within the program.

[4] http://www.numpy.org/
[5] http://www.scipy.org/
[6] http://www.netlib.org/lapack/
[7] https://www.gnu.org/software/gsl/

Developers often neglect accuracy errors requiring precision enhancement, as only a minimal subset of inputs triggers these errors [36], and precision augmentation significantly degrades computational efficiency. For instance, NumPy issue #1063 was closed without fixing. Interestingly, developers can face uncertainty in determining whether an error can be resolved within original precision constraints. The SciPy issue #6368 exemplifies this phenomenon: developers initially attempted to address the error by modifying the computation order, but ultimately determined that extended-precision floating-point arithmetic was necessary to achieve correct results. Furthermore, developers tend to employ the Remez algorithm[8], which requires high-precision programs to obtain oracles and conduct approximations, to perform a repair. Therefore, we aim to design a method capable of identifying and repairing original-precision-repairable errors without relying on high-precision floating-point arithmetic.

3.2 Original-precision-repairable errors

Certain instances of errors can be significantly mitigated through the application of series expansion without using high-precision arithmetic. Consider the function $\sin(x+\epsilon) - \sin(x)$, where the values of x and ϵ are 2.13 and 1×10^{-6}, respectively. We evaluate this function using the C++ Mathematical Library with double-precision arithmetic. For high-precision computation implementation, we utilize the *mpmath* [14]. Since the values of $\sin(x+\epsilon)$ and $\sin(x)$ are close to each other, the relative error value is 1.1095×10^{-10}, demonstrating the cancellation error. According to the Taylor series (cf. Section 2), we take $\sin(x+\epsilon)$ as the point of sin near x and have: $\sin(x+\epsilon) = \sin(x) + \frac{\cos(x)}{1!}(\epsilon) + \frac{-\sin(x)}{2!}(\epsilon)^2 + \frac{-\cos(x)}{3!}(\epsilon)^3 + \frac{\sin(x)}{4!}(\epsilon)^4 + \frac{\cos(x)}{5!}(\epsilon)^5 \cdots$. Interestingly, by omitting the $\sin(x)$ term and calculating the remaining terms of the Taylor series directly, we can avoid the cancellation and mitigate the floating-point error of the target function. This approach achieves a relative error of 1.6176×10^{-16}, illustrating a six-order-of-magnitude improvement in accuracy. We define this type of error as **original-precision-repairable errors**.

3.3 Limitations of existing repair methods

Floating-point errors are inevitable and can result in substantial inaccuracies with potentially harmful real-world consequences; therefore, many tools have been proposed to repair these errors. However, the existing tools cannot detect and repair the aforementioned repairable floating-point errors effectively.

Limitation 1: Inability to repair floating-point errors without transitioning entire floating-point programs to high-precision versions. Based on the observation that large floating-point errors tend to be triggered by inputs localized in small input intervals, AutoRNP [30] selects some points and corresponding oracles in the intervals as samples and utilizes linear approximation to construct patches. To obtain oracles of the sample points, the authors transform the complete program into a high-precision implementation. However, the transform processes require much expertise and effort, because directly increasing numerical precision (e.g., replacing double-precision (64-bit) floating-point numbers with 128-bit extended-precision format) can be ineffective and even introduce substantial errors [25]. For example, Figure 3 illustrates a simplified code piece in exp function from GLIBC, which aims to round x to an integer [25]. It is noted that this function only works for double-precision floating-point numbers. If we execute this function using higher-precision data types (such as long double), we typically will not observe the rounding effects seen with double-precision types. This is because long double variables possess more bits in significand. Consequently, the computation will generally return the exact value of x, violating the original semantics. Therefore, converting the entire program into a correct high-precision version requires significant time and extensive domain knowledge. Moreover, the execution time of high-precision programs is prolonged. For instance, execution using quadruple precision (128 bits) is nearly two orders of magnitude slower than double precision (64 bits) [11].

```
1 double round(double x) {
2     double n = 6755399441055744.0;
3     return (x + n) - n;
4 }
```

This function is only compatible with programs that use double-precision arithmetic.

Figure 3: One precision-specific code snippet in GLIBC.

Limitation 2: Only capable of resolving a subset of the errors. A line of research improves the accuracy of floating-point programs by using mathematically equivalent transformations [4, 17, 26, 29]. For example, the value of $\sqrt{x+1}-\sqrt{x}$ is equal to $1/(\sqrt{x+1}+\sqrt{x})$ by rationalizing the numerator: $\sqrt{x+1}-\sqrt{x} = \frac{(\sqrt{x+1}-\sqrt{x})(\sqrt{x+1}+\sqrt{x})}{\sqrt{x+1}+\sqrt{x}} = \frac{(\sqrt{x+1})^2-(\sqrt{x})^2}{\sqrt{x+1}+\sqrt{x}} = \frac{1}{\sqrt{x+1}+\sqrt{x}}$. Consequently, this transformation can avoid the cancellation cases when x is approaching 0. As for our illustrative example, Herbie [17] converts the original express to $(\sin(5\times10^{-7})\times2.0)\times\cos((-5\times10^{-7}-x))$ by using trigonometric transformations. However, these approaches rely on predefined templates, limiting their ability to cover all possible repairable floating-point errors. For instance, Herbie only supports some basic mathematical functions, such as **exp**, **log**, and **sin**, but cannot handle more complex ones like those provided by the GNU Scientific Library (GSL) [9].

ACESO [34] is the only existing tool that performs both detection and repairing of floating-point errors without relying on high-precision floating-point arithmetic. They detect the erroneous operations by calculating atomic condition numbers [36]. Similar to AutoRNP, ACESO utilizes polynomial approximation to obtain patches. However, instead of calculating oracles of sample points by using high-precision programs, ACESO computes oracles by capturing the *micro-structure* characterization of floating-point errors. Specifically, it selects points near the erroneous input within a carefully determined radius and computes their average output, which approximates the oracle value. Despite its innovation, ACESO's repairing performance exhibits limitations in certain scenarios. For

[8] https://github.com/scipy/scipy/issues/4034

[9] https://www.gnu.org/software/gsl/

example, in our illustrative case, ACESO's patch achieves a relative error of magnitude 10^{-9}, which significantly falls short of the accuracy delivered by Herbie 10^{-16} and the Taylor series-based approach 10^{-16}.

Limitation 3: Failure to identify original-precision-repairable errors. Effective error repair fundamentally requires precise error detection and localization, a capability currently lacking in state-of-the-art tools. The most closely related approaches are template-based repair methods, which nevertheless exhibit limited coverage of such errors. Furthermore, they lack theoretical foundations to understand these reparable cases.

To address these limitations, we propose our method, named OFP-Repair, to detect and repair the original-precision-repairable errors systematically and effectively.

4 OFP-Repair: detect and repair original-precision-repairable errors

In this section, we elaborate on our method, OFP-Repair. We begin by presenting the theoretical foundations of our method, elucidating the design rationale. Regarding error detection, we evaluate whether the condition number of the input with respect to the program is sufficiently small. Errors meeting this criterion are classified as original-precision-repairable errors. Subsequently, we derive corrective patches through series expansion. We use the illustrative function $\sin(x+\epsilon) - \sin(x)$ in Section 3 to show our detection and repair process. An overview of our method is shown in Figure 4.

4.1 Foundations of OFP-Repair

Before introducing the details of our method, this subsection establishes the theoretical foundations of our method. We first analyze the origins of large floating-point errors, demonstrating that they predominantly arise from cancellations (cf. subsection 2.3). Second, we show that programs can be converted into polynomial representations using Weierstrass approximation theorem [24].

Significant errors stem from cancellations. Before repairing errors, we investigate the root causes of floating-point inaccuracies. Zou et al. [36] proposed that substantial floating-point errors originate from atomic operations (such as $+, \times$, sin, cos, and exp) within large condition numbers. However, at the hardware/implementation level, these operations are ultimately composed of basic arithmetic operations including addition, subtraction, multiplication, and division. Notably, for these operations which may lead to significant condition numbers, only addition and subtraction are among the fundamental hardware-supported arithmetic operations [10]. When the condition numbers of addition and subtraction are large, cancellations appear in the program. Therefore, this analysis consequently demonstrates that the source of significant floating-point errors can be traced to cancellation effects.

We take the sin function of the GNU Scientific Library as an example to show that large floating-point errors of the atomic operations are attributed to cancellations. In subsection 2.1, we introduce that the *gsl_sf_sin_e* function simulates sin by using Taylor series for inputs with small absolute values. Concerning large inputs, *gsl_sf_sin_e* first reduces the input to the principal domain $[-\pi, \pi]$ and determines its octant location. When computing the remainder term, cancellations occur for inputs that are integer multiples of π (excluding zero). Zou et al. [36] found the same domains that can lead to large condition numbers of sin by using mathematical derivation. Therefore, cancellations lead to the large floating-point errors of sin function.

Programs are convertible to polynomial representations. To begin with, we consider the following two well-known theorems:

Theorem 1 (Arithmetic Operations Preserve Continuity Theorem [23]): *If f and g are continuous at a and c is a constant, then the following functions are also continuous at a: $f+g$, $f-g$, cf, fg, and $\frac{f}{g}$ if $g(a) \neq 0$.*

Theorem 2 (Weierstrass Approximation Theorem [24]): *If $[a,b]$ is an interval, $f\colon [a,b] \to \mathbb{R}$ is a continuous function, and $\varepsilon > 0$, then there exists a polynomial P on $[a,b]$ such that $d_\infty(P,f) \leq \varepsilon$ (i.e., $|P(x) - f(x)| \leq \varepsilon$ for all $x \in [a,b]$).*

As previously discussed, arithmetic parts of floating-point programs can be fully decomposed at the low level into arithmetic operations (addition, subtraction, multiplication, and division) involving input variables and constants. Therefore, according to Theorem 1, any floating-point program is continuous within the domain of the corresponding branch. It is crucial to note that this continuity must be constrained within individual branches, as different branches may employ distinct expressions, thereby breaking continuity across branch boundaries. For example, *gsl_sf_sin_e* employs distinct computational procedures for small and large absolute values. Furthermore, as established by Theorem 2, the original floating-point program can be equivalently represented as a polynomial within each branch domain. Since distinct polynomials can be constructed for different branches, we can conclude that floating-point programs are reducible to polynomial representations in general.

A critical observation is that while certain cancellation errors can be effectively mitigated through careful manipulation of original-precision arithmetic, others prove fundamentally irreducible within the precision constraint. Therefore, we assume that original-precision-repairable programs can be converted into polynomial representations that are free of cancellation.

4.2 (Overcoming limitation 1 and 3) Step 1: Detection

In this subsection, we first present the design inspiration. Subsequently, we show the details to identify the original-precision-repairable errors.

Design inspiration: How to isolate original-precision-repairable errors from general floating-point errors? Prior to answering this question, we examine the mechanism by which ATOMU detects floating-point errors. In Section 2.3, we have demonstrated that the subtraction of two floating-point numbers with similar magnitudes results in significant errors from the floating-point arithmetic perspective. However, if the real numbers x and y in Figure 2 are exactly representable as floating-point numbers (i.e., the truncated portions denoted by ellipses are all zeros), then the error becomes zero. Therefore, cancellation is highly probable, though not guaranteed, to produce substantial floating-point errors. To measure the magnitude of probability, ATOMU utilizes the condition number theory:

$$\left|\frac{f(x+\Delta x) - f(x)}{f(x)}\right| \approx \left|\frac{\Delta x}{x}\right| \cdot \left|\frac{x f'(x)}{f(x)}\right| \tag{3}$$

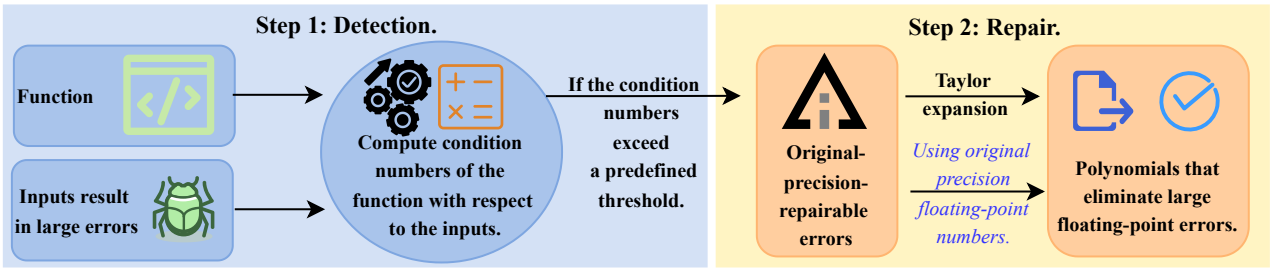

Figure 4: An overview of OFP-Repair.

where Δx denotes a small perturbation. $\left|\frac{f(x+\Delta x)-f(x)}{f(x)}\right|$ and $\left|\frac{\Delta x}{x}\right|$ mean relative errors of output and input, respectively. $\left|\frac{xf'(x)}{f(x)}\right|$ is the corresponding condition number, representing the error gain from input to output. For example, concerning our motivation example in Section 3, we take $\sin(x+\epsilon)$ and $\sin(x)$ as variables, respectively. The corresponding condition number with respect to $\sin(x+\epsilon)$ is $\left|\frac{\sin(x+\epsilon)}{\sin(x+\epsilon)-\sin(x)}\right|$, resulting in a value of 9.0132×10^9. Due to the inherent inaccuracy of floating-point arithmetic, $\sin(x+\epsilon)$ is likely to contain minor errors, which can result in significant error propagation in the final results. It should be noted that this amplification mechanism is mathematically derived—even if the subtraction operation itself is performed without errors, the inherent errors in the operands will lead to significant error in the final result. Given that ATOMU computes condition numbers for atomic operations (e.g., subtraction) to identify significant errors, why can we not instead compute the condition number of the entire function with respect to its inputs? Such an approach would enable us to determine the error amplification factor of input perturbations even under the scenario that all other components (e.g., operations) of the function execution are free of error. A small condition number implies that the function's mathematical formulation does not substantially amplify input errors. Consequently, if significant floating-point errors are observed, they can be mitigated by reformulating the function. Such errors are referred to as original-precision-repairable errors.

OFP-Repair: Identifying original-precision-repairable errors by computing the condition number of the entire program. Following ACESO [34], we first utilize ATOMU [36] to find substantial floating-point errors. Subsequently, for each significant error, we compute the condition number of the input with respect to the function. If the condition number is below a certain threshold, we consider the error to be original-precision-repairable.

With reference to our example in Section 3, the condition number with respect to x is $\left|\frac{x(\cos(x+\epsilon)-\cos(x))}{\sin(x+\epsilon)-\sin(x)}\right|$, leading to a value of 3.4034179**17418655**. Since it is difficult to obtain derivatives of some functions, we use the numerical differentiation approach to estimate the derivative: $f'(x) \approx (f(x+h)-f(x))/h$, resulting in a value of 3.4034179**5514549854**. As our focus is on the order of magnitude of the condition number rather than precise accuracy, the error introduced by the finite difference method is negligible for our purpose. It is noted that the condition number is 9.0132×10^9 if we take the $\sin(x+\epsilon)$ as the variable. Therefore, the small condition number we compute indicates that it is possible to mitigate the detected significant error using the original input x, without increasing the floating-point precision.

Our method enables developers to assess whether the SciPy issue #6368 (cf. subsection 3.1) can be repaired by using original precision floating-point numbers. By using OFP-Repair, the condition numbers respect to the two operands are about 2×10^{30} and 2×10^5, respectively. Therefore, we can confirm that this bug necessitates the implementation of high-precision floating-point arithmetic for repair.

Since OFP-Repair does not utilize high-precision programs and identifies the original-precision-repairable errors successfully, we can overcome limitation 1 and 3.

4.3 (Mitigating limitation 1 and 2) Step 2: Repair

This subsection first introduces the design rationale, followed by the procedure used for repair.

Design rationale: How can the original-precision-repairable program be transformed into one that eliminates the primitive cancellation? We have shown that significant floating-point errors stem from cancellations (cf. Subsection 4.1) and our goal of repair is to avoid these cancellations. We also demonstrate that the floating-point program can be transformed into polynomials with respect to the inputs. Therefore, we aim to transform original-precision-repairable programs into polynomial representations that are free of cancellations.

OFP-Repair: Repairing original-precision-repairable errors by using series expansions. We utilize Taylor series to transform the original programs into polynomials. This choice is motivated by three primary reasons. First, the erroneous inputs lie within a small neighborhood [36], where the Taylor series is used to estimate functions near one point. Second, the Taylor series converge rapidly [23], making it computationally efficient. Finally, the terms in the Taylor expansion differ significantly in magnitude, which prevents cancellation from occurring (if the program is original-precision-repairable). Consequently, the primitive function is transformed into: $\frac{\cos(x)}{1!}(\epsilon)+\frac{-\sin(x)}{2!}(\epsilon)^2+\frac{-\cos(x)}{3!}(\epsilon)^3+\frac{\sin(x)}{4!}(\epsilon)^4+\frac{\cos(x)}{5!}(\epsilon)^5\cdots$, which avoids the primitive cancellation and mitigates the error.

Since our repair process does not rely on high-precision programs and can be generalized to many cases based on our theoretical analysis and assumption, OFP-Repair can mitigate limitation 1 and 2. In the following section, we will evaluate the generality.

The limitation of our method: The Taylor expansion requires the function to be convergent at the expansion point. Consequently, our method becomes inapplicable when the function diverges at

the expansion point. For instance, as demonstrated in SciPy issue #3545, the term $norm.ppf(1-q/2)$ diverges as q approaches zero.

5 Evaluation

In Section 4, we demonstrate that OFP-Repair can address the three limitations of existing methods to repair original-precision-repairable errors: 1) Either the detection or the repair part requires high-precision programs; 2) Can only address a fraction of such errors; 3) Inability to identify such errors. In this section, we perform empirical evaluations to assess the effectiveness of our method in detecting and repairing the original-precision-repairable errors.

Our evaluation begins with a detailed presentation of the experimental setup, covering the datasets, evaluation metric, and our experimental environment. Subsequently, we present the experimental results to illustrate that our method can successfully detect and repair the original-precision-repairable errors. Since ACESO [34] is currently the only existing tool that does not require high-precision programs for repair, we select it as the baseline approach in our work.

5.1 Experiment setup

5.1.1 Dataset. Our evaluation is conducted using datasets from ACESO [34], which comprise 32 benchmark functions. Among these, 15 functions have been employed in prior floating-point error analysis and repair studies [17, 22, 26], where they were successfully corrected using original-precision floating-point numbers. The remaining 17 functions are variants of original-precision-repairable functions of the literature [9] that incorporate library calls from GSL and ALGLIB, making it challenging to construct high-precision versions for them. Consequently, oracle-based repair methods [30] are difficult to deploy on these functions. Additionally, due to the presence of external function calls, rewriting-based approaches [17] are also less applicable. Overall, all 32 functions can be repaired using original-precision floating-point numbers. By analyzing the bugs and patches collected by Franco et al. [5], we identify five original-precision-repairable errors. We use these cases to assess the performance of our approach on real-world bugs.

Although our method does not require high-precision oracles during either error detection or repair, we leverage the high-precision programs of ACESO as ground truth to validate the error reduction capability of our patches. For real-world cases, we adopt the patches provided by the developers.

5.1.2 Evaluation metric. We utilize relative error to measure floating-point errors: $\left|\frac{Result_{\text{approximate}}-Result_{\text{true}}}{Result_{\text{true}}}\right|$, where $Result_{\text{approximate}}$ represents for the results of patches and $Result_{\text{true}}$ denotes the ground truth from high-precision programs.

5.1.3 Software and hardware environment. Our experimental environment is a Docker container running Ubuntu 24.04 on a system with a 13th Gen Intel(R) Core(TM) i9-13900K CPU and 128GB RAM.

5.2 Evaluation results

This subsection presents the experimental results of evaluating our proposed method, OFP-Repair, structured around three key research questions (RQs). For each RQ, we first outline its motivation and the corresponding evaluation methodology, then provide a comprehensive analysis of the results.

RQ1 (addressing limitation 1 and 3): Can OFP-Repair identify original-precision-repairable errors?

Motivation. OFP-Repair employs numerical differentiation to compute program derivatives with respect to inputs, which are then used to compute condition numbers. When an input produces significant floating-point errors but exhibits a small corresponding condition number, we classify such errors as original-precision-repairable. To evaluate our method's capability in detecting these repairable errors, we implement OFP-Repair on all 32 functions from ACESO. It is noted that these 32 cases are original-precision-repairable. Since numerical differentiation may yield derivatives with significant errors, we conduct experiments to demonstrate that these inaccuracies do not compromise the method's effectiveness in identifying original-precision-repairable errors.

Approach. The datasets of ACESO [34] comprise 32 functions along with narrow input ranges that induce significant errors. The midpoint of each range yields the maximum error, indicating that all points within these ranges result in extremely large condition numbers for certain operations in the program. For instance, in Section 4.2, our illustrative example demonstrates a corresponding condition number of 9.0132×10^9. In ATOMU [36], a condition number of 1×10^5 is considered exceptionally large. To approach the midpoint and trigger a significant error of the function result, we compute the condition number for points at a distance with a radius of 1×10^{-5} and examine whether its corresponding condition number is significantly smaller than 1×10^5. Subsequently, we compare the analytical solution derived using derivative rules to assess whether the error in our numerical derivative computation affects the detection performance of our method.

Results. **OFP-Repair effectively identifies the original-precision-repairable errors.** Among these 32 functions, the maximum, minimum, and average values of the computed condition numbers are 1.47, 0, and 0.31, respectively. All computed values are significantly smaller than 1×10^5. Furthermore, since our sampling points are positioned extremely close to range midpoints, the condition numbers of operations susceptible to cancellation should theoretically be very large (exceeding 1×10^5). Therefore, the computed condition numbers of input-to-function mappings are sufficiently small to reliably identify these original-precision-repairable errors.

The precision of numerical derivatives has negligible impact on OFP-Repair's detection effectiveness. Table 2 demonstrates the results of our numerical derivatives and the corresponding relative errors. The largest relative error occurs with bj_tan, as its analytical derivative equals 0—theoretically suggesting an extremely small condition number at this point. Excluding this case, the observed relative errors range from 0 to 0.746. These errors do not affect our condition number estimation, as only order-of-magnitude precision is required rather than exact values. Consequently, numerical derivative errors have no material impact on our method's identification accuracy.

Table 2: Numerical derivatives and relative errors compared to analytical solutions. (Note: We set the relative error of bj_tan to 1 because the analytical value is 0. For cos_x2, the relative error is 0 because both analytical and numerical derivatives are 0.

Function Name	Numerical Derivative	Relative Error	Function Name	Numerical Derivative	Relative Error	Function Name	Numerical Derivative	Relative Error	Function Name	Numerical Derivative	Relative Error
exp_BI	1.25E-01	8.88E-06	Si_tan	-8.47E-02	5.00E-01	fc_bj	5.00E-01	1.79E-06	exp_1	1.00E+00	5.00E-06
bJ_sin	2.50E-01	2.39E-06	by_psi	2.62E+00	1.66E-01	cos_x2	0.00E+00	**0.00E+00**	x_tan	5.95E-01	4.34E-06
di_tan	2.34E-01	2.09E-02	fdm_log	2.50E-01	1.59E-06	exp_2	1.00E+00	1.67E-06	log_log	-1.00E+00	5.56E-06
log_erf	-1.20E+00	2.94E-06	eQ_sqrt	1.25E-01	1.32E-06	cos_sin	5.00E-01	0.00E+00	log_x	-2.00E+00	1.31E-10
acos_fd	2.50E-01	2.81E-05	W_var	-4.60E-02	1.27E-05	sin_sin	-1.55E-11	4.76E-01	sqrt_exp	3.54E-01	3.81E-06
ei	8.86E-01	9.48E-06	W_log	-3.07E-02	1.87E-05	tan_tan	3.10E-11	4.76E-01	sin_tan	1.91E-01	7.46E-01
Q1_W	2.00E+00	3.20E-06	pow_df	-1.36E+00	5.03E-06	cos_cos	-1.00E-06	1.10E-05	exp_x	5.00E-01	2.22E-06
bj_tan	-1.67E-06	**1.00E+00**	chi_ci	-8.88E-01	5.00E-01	exp_exp	2.00E+00	0.00E+00	x_x2	-1.00E+00	1.00E-05

Our method effectively detects original-precision-repairable errors without high-precision programs. This capability directly addresses limitations 1 and 3 of existing tools (as outlined in Section 3).

RQ2 (addressing limitation 1 and 2): Can OFP-Repair repair original-precision-repairable errors?

Motivation. OFP-Repair utilizes Taylor series to convert the programs into polynomials to obtain patches. To validate the repair effectiveness of our method, we conduct comparative experiments against ACESO, demonstrating that OFP-Repair not only enhances the original program's accuracy but also outperforms ACESO in error correction.

Approach. For a rigorous comparative evaluation between the patches generated by our method and those produced by ACESO, we adopt identical experimental procedures to those used in ACESO's implementation. The ACESO dataset comprises 32 functions along with input ranges that induce large errors in these functions. The radius of these ranges is 0.01, with the midpoint corresponding to the point of maximum error. Their evaluation approach involves sampling points within these ranges and computing their corresponding errors, ultimately selecting the maximum value as the final result (where lower maximum values indicate better repair performance). Two distinct sampling strategies were employed: (1) focused sampling near range midpoints which can lead to extremely substantial errors ('on Decayed Area'), and (2) uniform sampling across the entire range ('on Stable Area'). In our experiments, at most ten terms of the Taylor series are retained, because the subsequent term can be canceled out. The maximum interval is 0.01, and the 10th power of 0.01 is 1×10^{-20}. In double-precision floating-point arithmetic, adding 1×10^{-20} to 0.01 still results in 0.01 due to finite precision.

Results. **OFP-Repair successfully repair the original-precision-repairable errors.** Figure 5 illustrates our method consistently outperforms ACESO across all cases. Specifically, for the maximum absolute error on the stable area, maximum relative error on the stable area, maximum absolute error on the decayed area, and maximum relative error on the decayed area, the average values obtained by our method across all functions are 4.11×10^{-16}, 7.47×10^{-16}, 2.13×10^{-16}, and 3.73×10^{-15}, repectively. In comparison, ACESO yields 2.45×10^{-13}, 2.74×10^{-9}, 2.45×10^{-13}, and 5.74×10^{-07} for the same metrics. Our method shows precision improvements over ACESO by three, seven, three, and eight orders of magnitude, respectively.

OFP-Repair exhibits robust error mitigation, maintaining consistent effectiveness regardless of the function type. Since the magnitude of absolute errors scales with the result values, we focus our analysis on relative errors. Across both stable and decayed regions, our method reduces the relative errors to approximately 10^{-16} for the majority of functions (with the exception of *W_log* and *sin_sin*). In contrast, ACESO exhibits unstable performance, with relative errors exceeding 10^{-10} for certain functions. Stable repair is critical in practice, as the absence of ground-truth oracles makes it impossible to empirically validate patching effectiveness. Unstable methods risk leading developers to adopt low-quality patches, which may consequently induce severe operational faults.

Our method repairs floating-point errors while maintaining original-precision across all target functions, eliminating the need for high-precision implementations. This advancement specifically overcomes limitations 1 and 2 (cf. Section 3) inherent in current fixing approaches.

RQ3 Can OFP-Repair be applied to errors in real-world projects?

Motivation. In RQ1 and RQ2, we demonstrate that our method performs well in both detection and repair on the ACESO dataset. Although this dataset has been widely used in prior research [17, 22, 26], the errors it contains do not originate from real-world software projects. To further validate the effectiveness of our approach, we apply our method to five original-precision-repairable errors collected by Franco et al. [5] from real-world programs.

Approach. We follow the settings and procedures outlined in RQ1 and RQ2. Among these five bugs, SciPy issue #4034, #3547, and #3545 correspond to 2, 1, and 2 cases, respectively. All of them exhibit significant errors when the input values approach zero. Therefore, we select zero as the midpoint and construct ranges with a radius of 0.01. We first compute the condition number for each function when the input is 1×10^{-5}. It should be noted that for the first case of SciPy issue #3545, the function value equals zero when the input is 1×10^{-5}. Consequently, we set the input to 0.1 instead. Subsequently, we employ our method to repair these five errors.

Results. **OFP-Repair successfully identify all the five cases.** The five computed condition numbers are 3.00, 0.41, 0.03, 5.43,

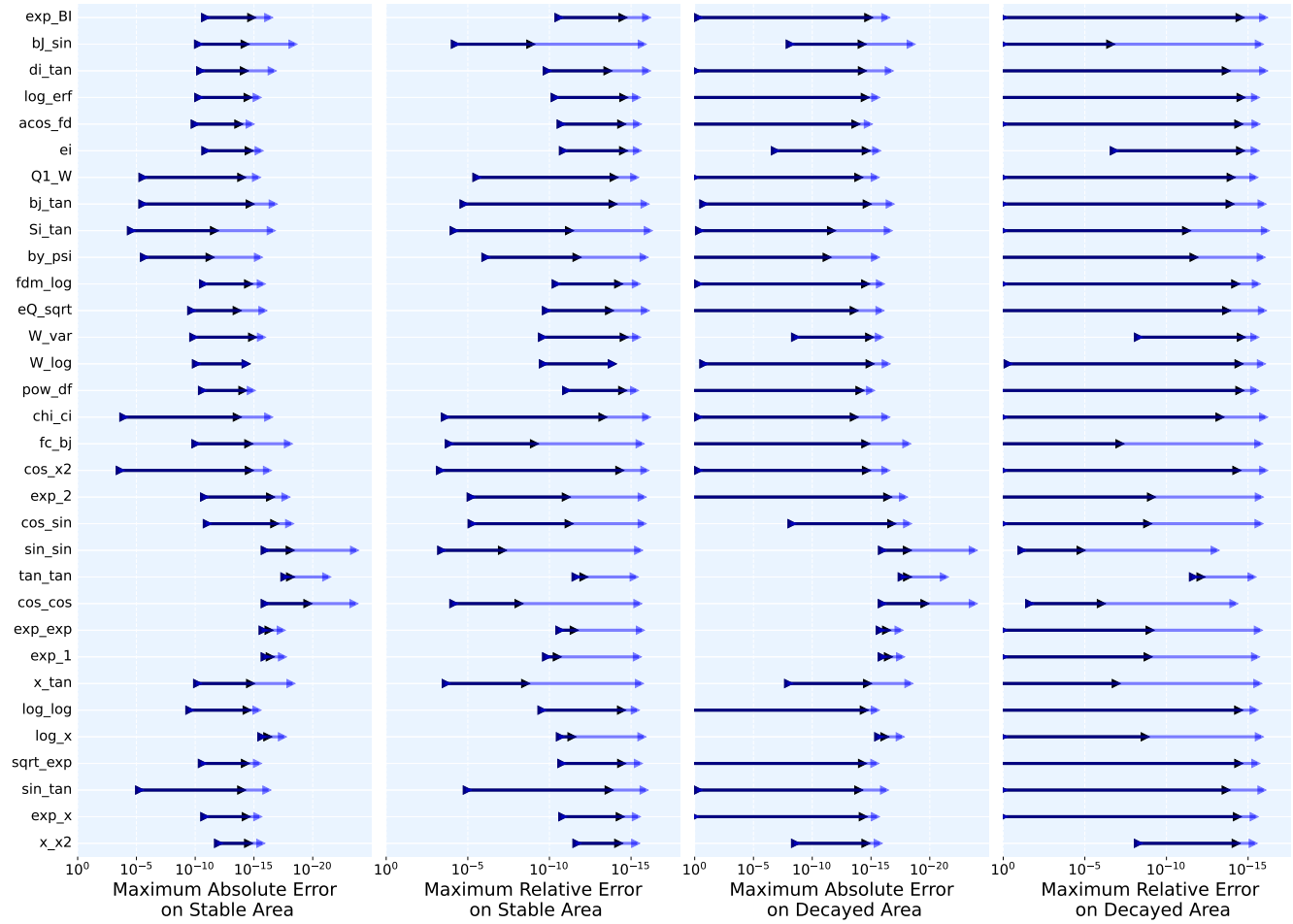

Figure 5: Accuracy improvements by ACESO and our method. Results are presented in terms of both absolute and relative errors, covering both the stable and decayed regions of the input ranges. The black arrowed lines originate from the maximum error of the original programs under our test inputs, with arrowheads indicating the maximum error after ACESO's fixing. Similarly, the blue arrowed lines depict the original maximum errors pointing to the errors improved by our proposed method.

and 0.03, all of which are significantly below 1×10^5. Thus, our method successfully identifies them as original-precision-repairable. Since developers may sometimes be uncertain whether a particular error necessitates high-precision fixes(cf. Section 3), OFP-Repair can provide them with a reliable determination.

OFP-Repair can successfully repair three of these bugs while ACESO can only fix one case. As shown in Figure 6, ACESO is only effective for the three subfigures corresponding to the first function. Moreover, the patches generated by our method achieve higher numerical accuracy. These results demonstrate the superior reliability of our method for repairing such floating-point errors in numerical software.

> Our method successfully detects all five original-precision-repairable errors. While the limitation of our Taylor expansion-based approach allows us to repair three functions, ACESO is only able to repair one function.

6 Case study: OFP-Repair on open bug reports

In Section 5, we show that our method can effectively detect the original-precision-repairable errors and repair them. However, the experiments conducted across our three research questions are all based on validated original-precision-repairable errors, which fail to fully demonstrate the distinctive advantages of our method. Therefore, in this section, we deploy our method to address bugs documented in an open report dating back over a decade [10]. This bug report contains 15 floating-point errors of GSL functions, and

[10] http://savannah.gnu.org/bugs/?43259

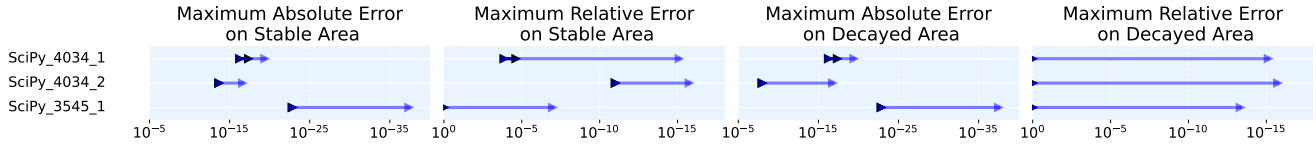

Figure 6: Performance comparison between ACESO and our method on three real-world cases.

our method successfully identifies and provides precision improvements by using original-precision arithmetic for five of these bugs:
1) Case 1: Fully resolved: The first example illustrates cases where our method can both identify the original-precision-repairable bugs and enhance precision using original-precision floating-point numbers to achieve a relative error below 1×10^{-16}. Three of the five bugs we can repair fall into this category.

$gsl\_sf\_hyperg\_0F1(c, x)$ results in a relative error of 1.15×10^{198} when the two operands are 3.39×10^{-215} and 3.95×10^{-242}. However, the condition numbers associated with the two parameters are 3.39×10^{-210} and 1.01×10^{-225}, respectively. Therefore, we identify the bug as the original-precision-repairable case. When c and x approach zero, the first two terms in the Taylor expansion of the function are $1 + x/a$. We use this as a patch, and the resulting relative error is only 1.17×10^{-27}.
2) Case 2: Partially addressed: This category contains two bugs, indicating that our method can identify cases where the accuracy can be improved using original-precision floating-point arithmetic. However, the relative error can only be reduced to just under 1×10^{-10}. For the first bug, $gsl\_sf\_gamma\_inc\_Q(a, x)$ leads to a significant error if the operands are 4.57×10^{-13} and 5.13×10^{-61}. Although we can confirm that this bug can be fixed using original-precision floating-point computation, the function is undefined at zero, rendering our Taylor expansion repair inapplicable (cf. Section 4.3). However, a part of the function can be well-approximated using a Taylor expansion. We apply our repair strategy to this part, reducing the relative error from 1.60×10^{-6} to 1.57×10^{-7}. Concerning the second bug, $gsl\_sf\_ellint\_P(\phi, k, n)$ produces a relative error as large as 9.52×10^{6}. Although the largest of the three condition numbers is 3.28×10^{5}, we believe that such a condition number should not theoretically result in an error of this magnitude. Upon inspecting the source code, we identify the root cause: the original implementation computes $\sin(\phi)^3$ directly, but due to ϕ being extremely small, the result underflows to zero. To improve accuracy, we modified the computation by separating the expression—multiplying $\sin(\phi)$ with a large variable first before cubing. This change successfully reduces the relative error to 1.91×10^{-9}.

7 Threats to Validity

In this section, we examine the potential threats to the validity of our study.
External validity. One potential threat is that the effectiveness of our method may be limited to specific types of data. To address this concern, we conduct evaluations on three distinct datasets: a widely adopted academic benchmark (RQ1 and RQ2), real-world bugs from practical projects (RQ3), and floating-point errors identified in open bug reports (cf. Section 6). Experimental results demonstrate that our method performs well across all datasets.
Internal validity. In our experiment, we set the upper limit of Taylor expansion terms to 10. A potential threat to internal validity is that if developers prioritize efficiency over precision in certain scenarios and reduce the number of terms, the accuracy of our generated patches may decrease. However, our configuration is justified based on the characteristics of our dataset. Specifically, our data requires that the input range of the patch spans a maximum distance of 0.01 from the point of highest error—a relatively large interval that necessitates a higher number of terms to maintain precision. In cases where the distance is significantly smaller—for instance, in $gsl\_sf\_sin\_e$ (cf. subsection 2.1), where the absolute value of the input is constrained to 1.22e-04—only two expansion terms are sufficient to achieve the desired accuracy.
Construct validity. In Section 5, our experiments only demonstrate that our detection achieves a high true positive rate; they do not provide evidence that it yields a high true negative rate. Obtaining ground truth for bugs that can only be fixed using high-precision arithmetic is challenging, as developers typically do not provide explicit clarification. Such bugs are often left open indefinitely or closed without resolution. However, in Section 6, we apply our method to bugs reported in open bug reports. We successfully identify and repair five bugs whose accuracy can be improved using original-precision floating-point arithmetic. This indicates that our method demonstrates a high true negative rate—at least higher than manual judgment. After all, if developers had recognized that these bugs could be fixed with original-precision arithmetic, they would likely have done so rather than leaving them unresolved for over one decade.

8 Related Work

This section outlines related research directions in error detection and repair.
Detecting errors of floating-point programs. Valgrind [15] is a dynamic binary instrumentation framework designed to support the construction of heavyweight dynamic analysis tools. Several techniques have been developed on top of Valgrind, including FPDebug [2], Verrou [7] and Herbgrind [20]. FPDebug employs randomized testing and shadow execution with higher-precision variables to detect catastrophic cancellations and rounding errors. Verrou leverages a Monte Carlo–based approach to identify errors. Herbgrind is designed to trace the root causes of numerical errors in large-scale programs by performing random sampling. Recently, several approaches have been introduced to identify inputs that cause significant floating-point errors [3, 8, 27, 30–33, 35, 36]. The motivation behind these methods lies in the observation that only a small subset of inputs typically leads to large errors in a program. Therefore, identifying such inputs is crucial. To this end, various techniques have been explored, including genetic algorithms, condition number analysis, and symbolic execution.

Repairing floating-point errors. Several studies have aimed to enhance the precision of floating-point computations by applying mathematically equivalent code transformations [4, 17, 26, 29]. However, these methods rely on high-precision computations to obtain an oracle that guides the transformations. Moreover, they are not capable of identifying all bugs that can be repaired using original-precision floating-point arithmetic. AutoRNP [30] uses high-precision programs to obtain oracles and employs linear approximation to estimate the correct function. To the best of our knowledge, ACESO [34] is the most closely related approach to ours. It does not rely on high-precision computation in either the detection or repair phase. However, ACESO performs poorly in many cases and cannot identify original-precision-repairable errors.

OFP-Repair aims to detect floating-point errors that can be repaired using original-precision arithmetic and to generate corresponding patches that operate within the same precision.

9 Conclusion

Floating-point errors can lead to severe consequences, among which certain types of errors can be repaired using original-precision arithmetic, while others necessitate high-precision solutions. Developers often refrain from addressing the latter due to the substantial resource demands of high-precision implementations. However, existing repair methodologies fail to assist developers in identifying the former category of errors, nor do they provide a unified framework for repair. To address this gap, we propose a novel method, named OFP-Repair. Experimental results demonstrate that our method effectively detects and repairs such errors. When compared to ACESO as a baseline repair algorithm, our patches exhibit significantly superior accuracy. Notably, OFP-Repair successfully resolves five out of 15 bugs in an open bug report.

In future work, we aim to extend the deployment of our method to a broader range of unresolved errors.

References

[1] Earl T Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. *ACM Sigplan Notices* 48, 1 (2013), 549–560.

[2] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A dynamic program analysis to find floating-point accuracy problems. *ACM SIGPLAN Notices* 47, 6 (2012), 453–462.

[3] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 43–52.

[4] Nasrine Damouche and Matthieu Martel. 2017. Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs. In *Automated Formal Methods*, Vol. 5. 63–76.

[5] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *32nd IEEE/ACM International Conference on Automated Software Engineering*. 509–519.

[6] Wade H Foy. 1976. Position-location solutions by Taylor-series estimation. *IEEE transactions on aerospace and electronic systems* 2 (1976), 187–194.

[7] Févotte François and Lathuilière Bruno. 2016. VERROU: a CESTAC evaluation without recompilation. In *17th International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerics (SCAN 2016)*.

[8] Hui Guo and Cindy Rubio-González. 2020. Efficient generation of error-inducing floating-point inputs via symbolic execution. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1261–1272.

[9] Richard Hamming. 2012. *Numerical methods for scientists and engineers*. Courier Corporation.

[10] Aboul-Ella Hassanien, Ajith Abraham, and Francisco Herrera. 2009. *Foundations of Computational Intelligence Volume 2: Approximate Reasoning*. Vol. 202. Springer Science & Business Media.

[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/.

[12] Jacques-Louis Lions et al. 1996. Flight 501 failure. *Report by the Inquiry Board* 190 (1996).

[13] Chenghu Ma, Liqian Chen, Xin Yi, Guangsheng Fan, and Ji Wang. 2022. NuMFUZZ: A Floating-Point Format Aware Fuzzer for Numerical Programs. In *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*. 338–347.

[14] The mpmath development team. 2023. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 1.3.0)*. `http://mpmath.org/`.

[15] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[16] Michael L Overton. 2001. *Numerical computing with IEEE floating point arithmetic*. SIAM.

[17] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. *Acm Sigplan Notices* 50, 6 (2015), 1–11.

[18] Jack Poulson, Bryan Marker, Robert A Van de Geijn, Jeff R Hammond, and Nichols A Romero. 2013. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Software* 39, 2 (2013), 1–24.

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

[20] 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*. 256–269.

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

[22] Alexey Solovyev, Marek S Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 1 (2018), 1–39.

[23] James Stewart. 2012. *Calculus: early transcendentals*. Cengage learning.

[24] Terence Tao et al. 2016. *Analysis II*. Springer.

[25] 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 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 619–630.

[26] Xie Wang, Huaijin Wang, Zhendong Su, Enyi Tang, Xin Chen, Weijun Shen, Zhenyu Chen, Linzhang Wang, Xianpei Zhang, and Xuandong Li. 2019. Global optimization of numerical programs via prioritized stochastic algebraic transformations. In *IEEE/ACM 41st International Conference on Software Engineering*. 1131–1141.

[27] Zheng Wang, Xin Yi, Hengbiao Yu, and Banghu Yin. 2022. Detecting High Floating-Point Errors via Ranking Analysis. In *2022 29th Asia-Pacific Software Engineering Conference*. 397–406.

[28] Eric W Weisstein. 1999. Roundoff error. *https://mathworld. wolfram. com/* (1999).

[29] Jinchen Xu, Mengqi Cui, Fei Li, Zuoyan Zhang, Hongru Yang, Bei Zhou, and Jie Zhao. 2024. Arfa: An Agile Regime-Based Floating-Point Optimization Approach for Rounding Errors. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1516–1528.

[30] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient automated repair of high floating-point errors in numerical libraries. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[31] Xin Yi, Hengbiao Yu, Liqian Chen, Xiaoguang Mao, and Ji Wang. 2024. FPCC: Detecting Floating-Point Errors via Chain Conditions. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 1504–1531.

[32] Zuoyan Zhang, Jinchen Xu, Jiangwei Hao, Yang Qu, Haotian He, and Bei Zhou. 2024. Hierarchical search algorithm for error detection in floating-point arithmetic expressions. *The Journal of Supercomputing* 80, 1 (2024), 1183–1205.

[33] Zuoyan Zhang, Bei Zhou, Jiangwei Hao, Hongru Yang, Mengqi Cui, Yuchang Zhou, Guanghui Song, Fei Li, Jinchen Xu, and Jie Zhao. 2023. Eiffel: Inferring Input Ranges of Significant Floating-point Errors via Polynomial Extrapolation. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering*. 1441–1453.

[34] Daming Zou, Yuchen Gu, Yuanfeng Shi, MingZhe Wang, Yingfei Xiong, and Zhendong Su. 2022. Oracle-free repair synthesis for floating-point programs. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 957–985.

[35] Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. 2015. A genetic algorithm for detecting significant floating-point inaccuracies. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 529–539.

[36] Daming Zou, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, Lu Zhang, and Zhendong Su. 2019. Detecting floating-point errors via atomic conditions. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–27.