Efficient Automated Repair of High Floating-Point Errors in Numerical Libraries

XIN YI, National University of Defense Technology, China
LIQIAN CHEN, National University of Defense Technology, China
XIAOGUANG MAO*, National University of Defense Technology, China
TAO JI, National University of Defense Technology, China

Floating point computation is by nature inexact, and numerical libraries that intensively involve floating-point computations may encounter high floating-point errors. Due to the wide use of numerical libraries, it is highly desired to reduce high floating-point errors in them. Using higher precision will degrade performance and may also introduce extra errors for certain precision-specific operations in numerical libraries. Using mathematical rewriting that mostly focuses on rearranging floating-point expressions or taking Taylor expansions may not fit for reducing high floating-point errors evoked by ill-conditioned problems that are in the nature of the mathematical feature of many numerical programs in numerical libraries.

In this paper, we propose a novel approach for efficient automated repair of high floating-point errors in numerical libraries. Our main idea is to make use of the mathematical feature of a numerical program for detecting and reducing high floating-point errors. The key components include a detecting method based on two algorithms for detecting high floating-point errors and a repair method for deriving an approximation of a mathematical function to generate patch to satisfy a given repair criterion. We implement our approach by constructing a new tool called AutoRNP. Our experiments are conducted on 20 numerical programs in GNU Scientific Library (GSL). Experimental results show that our approach can efficiently repair (with 100% accuracy over all randomly sampled points) high floating-point errors for 19 of the 20 numerical programs.

CCS Concepts: • **Mathematics of computing → Mathematical software**; **Computations in finite fields**; • **Theory of computation → Numeric approximation algorithms**; • **Software and its engineering →** *Search-based software engineering*;

Additional Key Words and Phrases: Floating-point errors, automated repair, numerical program

ACM Reference Format:
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 (January 2019), 29 pages. https://doi.org/10.1145/3290369

*Corresponding author

Authors' addresses: Xin Yi, Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, China, yixin09@nudt.edu.cn; Liqian Chen, Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, China, lqchen@nudt.edu.cn; Xiaoguang Mao, Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, China, xgmao@nudt.edu.cn; Tao Ji, Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, China, taoji@nudt.edu.cn.

2475-1421/2019/1-ART56
https://doi.org/10.1145/3290369

1 INTRODUCTION

Using floating-point representation instead of real arithmetic in numerical programs aims to make the calculation fast. However, the floating-point arithmetic accompanied with roundoff and truncation errors cannot guarantee that the results of numerical programs always have sufficient accuracy. Numerical programs in numerical libraries that intensively involve floating-point computations may encounter high floating-point errors. Hence, it is highly desired to reduce high floating-point errors in widely used numerical libraries.

One method to reduce high floating-point errors is to use higher precision to perform floating-point calculation of the original program. For example, one may replace a 32-bit single precision with a 64-bit double precision to improve the accuracy of results. However, higher precision execution will slow down the execution, sometimes even a thousand times slower [Benz et al. 2012]. In addition, higher precision execution may introduce extra errors and thus may not be able to improve the accuracy of numerical programs in numerical libraries which may involve certain precision-specific operations [Wang et al. 2016].

Mathematical rewriting is another choice, which reduces floating-point errors by rearranging floating-point expressions or taking Taylor expansions. Using this method does not need higher precision but requires users know the finer details of floating-point arithmetic. Along this direction, tools like Herbie [Panchekha et al. 2015] and Salsa [Damouche and Martel 2018] were developed to utilize mathematical rewriting to generate more accurate floating-point expressions automatically. However, the mathematical rewriting may also fail to find a more accurate expression within a limited search space constrained by the number of laws of mathematical transformation. In particular, the mathematical rewriting may not fit for reducing a high floating-point error evoked by an ill-conditioned problem (indicated by a large condition number, see §2) that is in the nature of the mathematical feature of many numerical programs in numerical libraries.

This paper aims to provide an efficient approach for automated repair of high floating-point errors in numerical libraries. To this end, several challenges need to be addressed. **The first challenge** is to detect high floating-point errors efficiently in numerical programs. The set of 64-bit floating point inputs of a numerical program is a huge search space, which makes the exhaustive search not practical. Moreover, numerical programs in numerical libraries are supposed to be already designed delicately for accuracy and inputs that can trigger the remaining high floating-point errors should localize in some small parts of the whole input domain, which makes the detecting more challenging. An efficient detecting method is desired to search for inputs that can trigger high floating-point errors in the huge search space. **The second challenge** which is also a key challenge is to reduce high floating-point errors to satisfy a given repair criterion. As mentioned before, using higher precision not only degrades performance, but also may introduce extra errors, while mathematical rewriting is constrained by limited search space and cannot handle the ill-conditioned problem. In particular, high floating-point errors evoked by ill-conditioned problem are hard to be repaired even for experienced developers of numerical libraries[1]. **The third challenge** is to reduce time overhead of repaired programs compared with the original programs. Performance is important for numerical libraries, and thus repaired programs should not introduce too much time overhead.

This paper addresses these challenges mainly by exploiting the mathematical feature of numerical programs. We suppose that a numerical program in numerical libraries is used to simulate a mathematical function. First, we make use of the condition number of the mathematical function and combine it with search algorithms to help detect high floating-point errors in the numerical

[1]For example, Di Franco et al. [2017] find a phenomenon that a bug (#6368: https://github.com/scipy/scipy/pull/6368) in Scipy was first repaired by mathematical rewriting, and developers recognized later that the bug is due to an ill-conditioned problem and it may still remain even after repair.

program (**the first challenge**). Then we directly extract an approximation based on the inputs and outputs of the mathematical function to generate a patch for repairing the numerical program to satisfy a given repair criterion (**the second challenge**). During the above process, we use a simple but efficient method to approximate the mathematical function and a search optimization to improve the performance of the generated patch, both of which are useful for reducing time overhead of repaired programs (**the third challenge**). The contributions of this paper are as follows:

- We present a detecting method that includes two novel algorithms (namely DEMC and PTB) for detecting high floating-point errors in numerical programs. More specifically, based on the guide of condition number, we use two global optimization algorithms (*the Differential Evolution algorithm* and *the Monte Carlo Markov Chain (MCMC) algorithm)* to help us find the input that can trigger the possible maximum floating-point error. As far as we know, there is no existing work combining condition number and MCMC for finding the input triggering maximum errors. Moreover, our detecting method not only finds the input that can trigger the maximum floating-point error (like existing detecting methods [Chiang et al. 2014] [Zou et al. 2015] [Yi et al. 2017b]), but also searches for inputs that can trigger floating-point errors higher than a given repair criterion. (See §4.1.)
- We present a repair method to produce patch automatically to reduce floating-point errors in a numerical program to satisfy a given repair criterion. We prove the guarantee of termination of our repair method for an arbitrary given repair criterion. Unlike existing methods that search for repairs by making changes of the implementation of the original numerical program, our method uses a piecewise quadratic function to approximate the corresponding mathematical function of the numerical program to generate patches. Therefore, the approximation is independent of the implementation of the numerical program and can be applied to a numerical program in other numerical libraries that simulates the same mathematical function. Moreover, our method uses a search optimization to improve the performance of a generated patch to reduce time overhead of repaired programs. (See §4.2 and §4.3.)
- We develop a prototype tool called AutoRNP and evaluate our approach by conducting experiments on 20 numerical programs of GSL. Experimental results show that our approach can efficiently repair high floating-point errors in 19 of 20 numerical programs of GSL (with 100% accuracy, i.e., after repair, 100% of our randomly sampled inputs yielding outputs that satisfy the given repair criterion). (See §5.)

The rest of the paper is organized as follows. We first introduce the basics of floating-point representation, the definition of high floating-point error and the ill-conditioned problem in §2. We then give an overview of our approach through an example in §3 and detail our approach in §4. We provide the details of implementation together with experimental results in §5. The limitations of our work are discussed in §6. In §7 we summarize related work. We end the paper with conclusions and future work (§8).

2 PRELIMINARIES

Floating-point representation According to the IEEE-754 Standard [Kahan 1996], a floating-point number can be represented in scientific notation:

$$f = (-1)^S \times M \times 2^E \tag{1}$$

where $S \in \{0, 1\}$ is the 1-bit *sign* of x, which represents that x is positive (when $S = 0$) or negative (when $S = 1$); $M = m_0.m_1m_2 \dots m_\mathrm{p}$ is called the *significand*, where $f = .m_1m_2 \dots m_\mathrm{p}$ represents a $\mathbf{p}$-bit fraction and m_0 is the hidden bit without need of storage; $E = e - \mathbf{bias}$ is called the *exponent*, where e is a biased **e**-bit unsigned integer and **bias** $= 2^{\mathbf{e}-1} - 1$. Taking the 64-bit double-precision format as an example, **e** $= 11$ (and thus **bias** $= 1023$), **p** $= 52$.

High floating-point error We give the definition of a high floating-point error as below:

Definition 1. *Let $f(x)$ represent a mathematical function, $f_p(x)$ represent the corresponding numerical program implementing $f(x)$. Given an error threshold ε, for an input x_0, if ErrorFunction$(f(x_0),\ f_p(x_0)) > \varepsilon$, we say the input x_0 triggers* a high floating-point error.

The *ErrorFunction* in Definition 1 is a function for measuring the floating-point error between the outputs of mathematical function and numerical program. In this paper, we define the floating-point error as the number of floating-point values between the mathematical function output O_r and numerical program output O_f, following [Panchekha et al. 2015]. This error can be represented by Eq. 2 and characterized by Eq. 3.

$$FPNum\{O_r,\ O_f\} = |\{a_i \in \mathbb{F} | min(O_r,\ O_f) \le a_i \le max(O_r,\ O_f)\}| \tag{2}$$

$$ErrBits\{O_r,\ O_f\} = log_2(FPNum\{O_r,\ O_f\}) \tag{3}$$

$FPNum\{O_r,\ O_f\}$ in Eq. 2 represents the number of floating-point values between the mathematical function output O_r and numerical program output O_f including themselves. $ErrBits\{O_r, O_f\}$ in Eq. 3 counts the number of most-significant bits that the approximate and the exact results agree on[2]. Compared with relative error and absolute error in real, $FPNum\{O_r, O_f\}$ and $ErrBits\{O_r, O_f\}$ can keep consistent over the input space and avoid special handling for infinite and denormalized values.

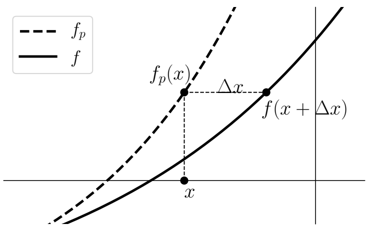


Fig. 1. Backward error

Ill-conditioned problem For a mathematical function $f(x)$, its condition number function can be expressed as follows:

$$C(x) = \left|\frac{f'(x) \cdot x}{f(x)}\right| \tag{4}$$

where $f'(x)$ denotes the derivative of the mathematical function $f(x)$.

Condition number is an important quantity for measuring how sensitive a function is to errors in the input. It has been mainly used to investigate instability of numerical programs [Bao and Zhang 2013] [Tang et al. 2017]. To illustrate the influence of condition number to floating-point error, we first introduce the notion of backward error. If we map the value given by a numerical program $f_p(x)$ to its corresponding mathematical function $f(x)$, as shown in Fig. 1, we can get

$$f_p(x) \simeq f(x + \triangle x) \tag{5}$$

We call $\triangle x$ the backward error B, and let $\delta = \triangle x / x$. We follow the assumption of [Fu et al. 2015] that the mathematical function f is smooth in a neighborhood of x and the backward error is small. Then by the Taylor expansion, we have

$$Forward\_error = \left|\frac{f(x) - f_p(x)}{f(x)}\right| = \left|\frac{f(x) - f(x + \delta \cdot x)}{f(x)}\right| \approx |\delta| \cdot \left|\frac{f'(x) \cdot x}{f(x)}\right| + \Theta(\delta^2) \tag{6}$$

[2]For example, $FPNum\{1.0,\ 2.0\} = 4503599627370497$ means there are 4503599627370497 number of floating-point values between 1.0 and 2.0 including themselves, and $ErrBits\{1.0, 2.0\} = log2(FPNum\{1.0, 2.0\}) = 52$ means that the number 2.0 has 52 bits error compared to 1.0. The value range of error threshold ε can be limited to $[1, 2^{64})$ by Eq. 2 and $[0, 64)$ by Eq. 3.

Eq. 6 shows that the forward error is mainly influenced by the backward error $B = |\delta|$ and the condition number $\left|\frac{f'(x) \cdot x}{f(x)}\right|$, i.e.,

$$Forward\_error \approx B \cdot C(x) \tag{7}$$

The ill-conditioned problem happens when the value of $C(x)$ is large. As shown in Eq. 7, if the value of $C(x)$ is large enough, even a small backward error B would lead to a large forward error. Note that the value of $C(x)$ is inherent in the mathematical function $f(x)$ (according to Eq. 4) while independent of the implementation of numerical program $f_p(x)$. Theoretically, it is very difficult to repair the high floating-point error evoked by the ill-conditioned problem.

Notations For mathematical function $f(x)$ and its corresponding numerical program $f_p(x)$, we use $X \subseteq \mathbb{F}$ to denote their input domain. We use the rounding function $fl : \mathbb{R} \to \mathbb{F}$ to convert a real number to a floating-point number. In this paper, we consider a floating-point input interval I which represents a set of floating-point numbers, i.e., $I = [x_0,\ x_n] = \{x_0,\ x_1,\ x_2, ...,\ x_n\}$. We use the default rounding mode "rounding to nearest even" of IEEE 754 standard in this paper. We use $\{+,\ -,\ \times,\ /\}$ to denote real-valued operations and $\{\oplus,\ \ominus,\ \otimes,\ \oslash\}$ to denote floating-point operations. We assume that each floating-point operation is left-associative. We use the same definition of *ulp* function (in double-precision) as [Lee et al. 2017] who follows the Goldberg's definition [Goldberg 1991]:

$$For\ r \in \mathbb{R},\ ulp(r) = \begin{cases} 2^{k-52} & if\ |r| \in [2^k, 2^{k+1})\ where\ k \in [-1022, 1023] \cap \mathbb{Z} \\ 2^{-1074} & if\ |r| \in [0, 2^{-1022}) \end{cases} \tag{8}$$

where *ulp* (unit in the last place) is the gap between the two floating-point numbers nearest to r, even if r is one of them. Formally, if a and b are two adjacent floating-point numbers around r satisfying $(r \geq 0.0 \wedge a \leq r < b) \vee (r < 0.0 \wedge a < r \leq b)$, then we have $ulp(r) = |a - b|$.

3 OVERVIEW

```
int gsl_sf_legendre_P3_e(double x, gsl_sf_result * result)
{
    result->val = 0.5*x*(5.0*x*x - 3.0);
    result->err = GSL_DBL_EPSILON * (fabs(result->val) + 0.5 * fabs(x) * (
        fabs(5.0*x*x) + 3.0));
    return GSL_SUCCESS;
}
```

Fig. 2. Code of gsl_sf_legendre_P3

In this section, we give an overview of our approach by illustrating how our approach repairs the high floating-point errors in a motivating example. The example is the program *gsl_sf_legendre_P3* in GSL. The source code of the *gsl_sf_legendre_P3* is shown in Fig. 2. The example implements the *Legendre functions* $P_n(x)$ with $n = 3$. The code in Fig. 2 shows that the result of the program is calculated by the polynomial "$0.5 * x * (5.0 * x * x - 3.0)$". The ill-conditioned problem exists around the roots of the polynomial. If letting $x = x_0 = 0.7745966692414834$, the polynomial returns output *zero*, while the mathematical output of *legendre_P3* at x_0 in real-number arithmetic should be *8.1726185204e-17*. In this case, the result of *FPNum*(*0*, *8.1726185204e-17*) is around *4.36611485515e+18* which implies around 61.9 bits error. More specifically, the expression $5.0 \otimes x \otimes x$ has a roundoff error *2e-16* for the input x_0, and the roundoff error is small for the value of $5.0 \otimes x \otimes x$ (which is rounded to 3.0), but huge for $5.0 \otimes x \otimes x \ominus 3.0$ (which is rounded to 0.0). Thus, the bad cancellation comes from the minus operation in the expression $5.0 \otimes x \otimes x \ominus 3.0$ which makes the roundoff error in $5.0 \otimes x \otimes x$ become a large relative error. The condition number at the input x_0 is around *7.341588237629298e+16*, so even the small rounding error introduced in $5.0 \otimes x \otimes x$ will be enlarged and propagated to the output due to the large condition number (according to Eq. 7).

The work-flow of our approach is shown in Fig. 3. We now introduce the repair process step by step.

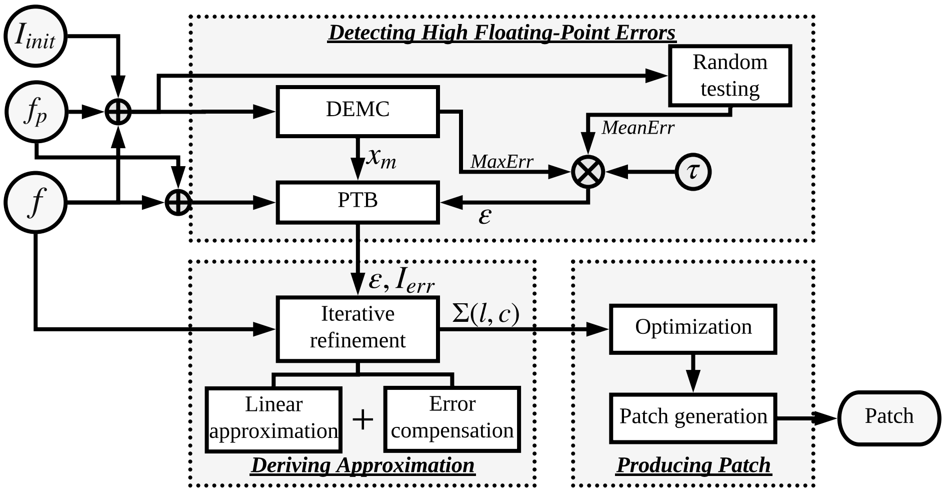
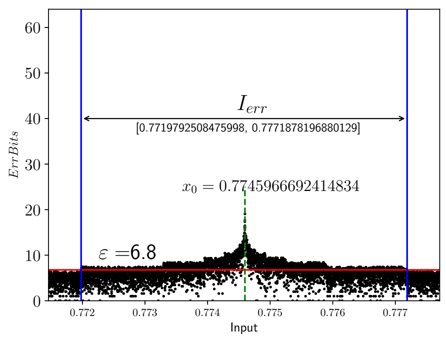


Fig. 3. Work-flow of our approach. $\boldsymbol{f_p}$: numerical program, $\boldsymbol{f}$: the corresponding mathematical function of f_p, $\boldsymbol{I_{init}}$: input domain of f_p, ***MeanErr***: the mean error of numerical program f_p in I_{init}, ***MaxErr***: the maximum error of numerical program f_p in I_{init}, $\boldsymbol{\tau}$: parameter given by users to adjust the error threshold in interval [*MeanErr*, *MaxErr*] (in Eq. 18)

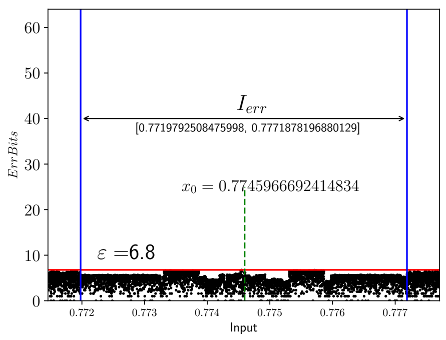
Detecting high floating-point errors. In this step, we try to find an input interval which includes inputs that can trigger floating-point errors higher than an error threshold ε. As shown in Fig. 3, first, we use the DEMC algorithm (§4.1) to search for an input x_m that can trigger a possible maximum floating-point error *MaxErr*. Then, we apply the PTB algorithm (§4.1) to find an input interval I_{err} which includes inputs that can trigger floating-point errors larger than the error threshold ε. Finally, the input interval I_{err} and error threshold ε will be passed to next step.

For the motivating example, we get the input $x_0 = 0.7745966692414834$ that can trigger a possible maximum floating-point error. Then, under the given threshold $\varepsilon = 6.8$, we get the input interval $I_{err} = [0.7719792508475998, 0.7771878196880129]$, as shown in Fig. 4(a).

Deriving an approximation of the mathematical function. After getting the input interval I_{err}, we will derive an approximation of the mathematical function f over I_{err} to satisfy the error threshold ε. We use a piecewise quadratic function (§4.2) to approximate the mathematical function over I_{err}. As shown in Fig. 3, we first use a linear function to approximate the mathematical function f, then use a quadratic function to compensate the error between the linear function and mathematical function f (§4.2) to make the approximation closer to the mathematical function f. Unfortunately, the approximation using only one linear function with error compensation of quadratic function may be not accurate enough to satisfy the error threshold, so we adopt an iterative refinement algorithm (§4.2) to iteratively use more pieces of linear approximation with error compensation to generate a piecewise quadratic function to approximate a mathematical



(a) Original *ErrBits* distribution



(b) *ErrBits* distribution after repair

Fig. 4. *ErrBits* distribution of program gsl_sf_legendre_P3

function over I_{err}. In the end of the process, we can get $\Sigma(l, c)$ which represents the piecewise quadratic function.

For the motivating example, under the error threshold $\varepsilon = 6.8$, a piecewise quadratic function with around 1000 pieces is produced after iterative refinement to approximate the mathematical function within the input interval I_{err} (including more than 4691 billion double-precision floating-point inputs) to satisfy the error threshold ε.

```
int gsl_sf_legendre_P3_e(double x, gsl_sf_result * result)
{
  if((x<=0.7771878196880129)&&(x>=0.7719792508475998)){
     result->val = accuracy_improve_patch_of_gsl_sf_legendre_P3(x);
     result->err = GSL_DBL_EPSILON * fabs(result->val);
     return GSL_SUCCESS;
  }
  result->val = 0.5*x*(5.0*x*x - 3.0);
  result->err = GSL_DBL_EPSILON * (fabs(result->val) + 0.5 * fabs(x) * (
       fabs(5.0*x*x) + 3.0));
  return GSL_SUCCESS;
}
```

Fig. 5. gsl_sf_legendre_P3 after repair

Producing patch. Finally, we convert the piecewise quadratic function $\Sigma(l, c)$ to patch. The piecewise quadratic function is implemented in the following manner: For an input x in the input interval I_{err}, we first find which piece of the piecewise quadratic function x belongs to, then we call the function representing the piece to compute the output for the input x. To reduce time overhead of searching for the right piece for an input, we apply a search optimization (§4.3) to accelerate the process of searching the piece of the piecewise quadratic function x belongs to. After optimization, we convert $\Sigma(l, c)$ with search optimization to a function in C code. We would like to induce as little as possible the influence of patch on the readability of original code, so we package the details of the piecewise quadratic function in a separate function. As a result, a patch includes two parts: 1) A function in C code which is stored in a separate file to implement the piecewise quadratic function; 2) A branch code fragment that is inserted into the source code of numerical program to decide whether to call the added function.

For the motivating example, the program after repair is shown in Fig. 5. From Fig. 5, we see that the patch of *gsl_sf_legendre_P3* is packaged in the function *accuracy_improve_patch_of_gsl_sf_legendre_P3*. The *ErrBits* distribution of the program after repair is shown in Fig. 4(b).

In summary, under a given error threshold, our approach localizes inputs that can trigger high floating-point errors, encloses them as a certain input interval, derives an approximation of mathematical function that satisfies the error threshold, and converts the approximation to a patch which is then inserted into source code to complete the repair.

Existing methods on the example To the best of our knowledge, no efficient way by mathematical rewriting can improve the accuracy of the polynomial "$0.5 * x * (5.0 * x * x - 3.0)$" around x_0. Factoring is a possible solution of mathematical rewriting to reduce errors around a root of a polynomial. When evaluating a polynomial $ploy(x) = (x \ominus r) \otimes Q(x)$ near root r , the execution of $x \ominus r$ in floating-point arithmetic is exact (according to Sterbenz's theorem [Sterbenz 1973], see §4.3). However, the factoring requires that the root r can be expressed exactly in floating-point number, otherwise, the small roundoff error of r will also lead to a large relative error (according to Eq. 7). In the example, the root of "$0.5 * x * (5.0 * x * x - 3.0)$" cannot be exactly expressed by a floating-point number, so the idea of factoring does not work for the example. Note that, the ill-conditioned nature of the problem also suggests that mathematical rewriting techniques (such as Herbie) will fail to reduce errors in the example[3].

Using higher precision can reduce errors around x_0 but will degrade the performance of the basic function *gsl_sf_legendre_P3* which is also called by other functions in GSL. For example, if we use 128-bit precision to calculate the polynomial "$0.5 * x * (5.0 * x * x - 3.0)$", the execution time is around 3.5 times slower than the original execution under 64-bit precision.

4 APPROACH

In this section, we details our approach in three parts which correspond to the work-flow in Fig. 3.

4.1 Detecting High Floating-Point Errors

Our detecting method is based on the following hypothesis:

HYPOTHESIS 1 (H1). *A numerical program in numerical libraries should result in outputs that are accurate enough for the most part of its valid input domain.*

The hypothesis is reasonable since numerical programs in a widely used and well maintained numerical library are supposed to be already designed delicately for accuracy and their outputs for most inputs are supposed to be with high accuracy. This hypothesis implies that high floating-point errors should be triggered by inputs located in some small input intervals of the valid input domain. Note that the input domain of a numerical program may be constrained by the feature of the (mathematical) function that the program implements and the value range of floating-point number[4], so we refer the input domain satisfying such constraints as the "valid" input domain. E.g., the valid input domain of exponential function *exp* is constrained around $[-709, 709]$ for 64-bit floating-point inputs. Note that we consider only valid input domain throughout this paper.

Based on the hypothesis, we propose a detecting method following the logic flow in Fig. 6. As shown in Fig. 6, first, we search for the input x_m that can trigger the possible maximum floating-point error $MaxErr^{(0)}$ in the whole input domain I_{init}. If $MaxErr^{(0)}$ is larger than error threshold ε,

[3]https://pavpanchekha.com/blog/float-point-polynomial.html
[4]E.g., the value range of double precision floating-point number is around $[-1.7e+308, +1.7e+308]$.

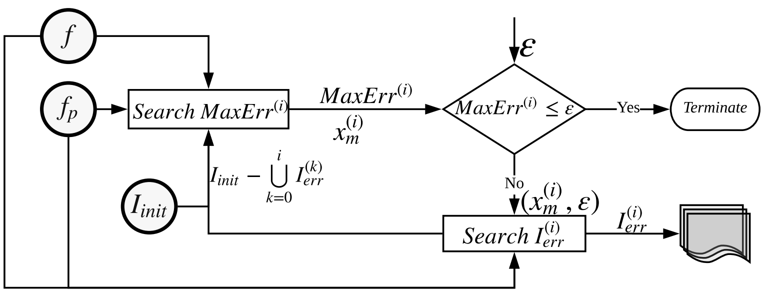


Fig. 6. Logic flow for detecting high floating-point errors. $\boldsymbol{f_p}$: numerical program, $\boldsymbol{f}$: corresponding mathematical function of f_p, $\boldsymbol{I_{init}}$: input domain of f_p, ***MaxErr***: the maximum error in $I_{init} - \bigcup_{k=0}^{i} I_{err}^{(k)}$, $\boldsymbol{\varepsilon}$: error threshold

then we search for an input interval $I_{err}^{(0)}$ around x_m that encloses inputs that can trigger floating-point errors higher than ε. The detecting method will terminate if the current $MaxErr^{(i)}$ found is less than ε, otherwise the method will iteratively find all $I_{err}^{(i)}$. Now, we introduce two algorithms to help us find $MaxErr^{(i)}$ and $I_{err}^{(i)}$.
Detecting the possible maximum floating-point error (MaxErr) We propose an algorithm called DEMC, based on two search algorithms *namely Differential Evolution algorithm* [Storn and Price 1997] and *Monte Carlo Markov Chain (MCMC) algorithm* [Andrieu et al. 2003], to search for an input that can trigger the possible maximum floating-point error. Differential evolution algorithm is a simple and efficient global optimization algorithm over continuous space. The algorithm operates on real numbers and naturally fits for numerical optimization. We use the algorithm to help us find the input triggering the possible maximum condition number over the whole input domain of a numerical program. The MCMC algorithm is a sampling method that draws samples from the target (usually unknown) distribution. MCMC has been used to search the maximum backward error [Fu et al. 2015] and to achieve high coverage for floating-point code [Fu and Su 2017], and has also been applied in STOKE [Schkufza et al. 2014] for stochastic search of floating-point optimization. We configure the MCMC sampling such that it tends to attain the inputs that may trigger maximum floating-point errors with higher probability than the other points. We use the MCMC algorithm to avoid local maxima and to find the input triggering maximum floating-point error in a relative smaller search space.

First, we formalize the problem of detecting the maximum floating-point error as follows:

Definition 2. *Let $f(x)$ represent a mathematical function, $f_p(x)$ represent the corresponding numerical program, $\mathbb{D} \in \mathbb{F}$ denote the valid input domain. The detecting problem is to search for an input $x_m \in \mathbb{D}$, such that $\forall x_i \in \mathbb{D}$, $ErrBits(f(x_m), f_p(x_m)) \geq ErrBits(f(x_i), f_p(x_i))$.*

Then, we use two fitness functions to guide our DEMC method to find such an input x_m.
Fit1: *ErrBits*. In this paper, we use *ErrBits* (defined in Eq. 3) to evaluate the error associated to an output of a numerical program. Hence *ErrBits* becomes a fitness function of our approach.
Fit2: *Appro*$(C(x))$. We use the approximate value of condition number $C(x)$ (Eq. 4) of the mathematical function $f(x)$ as our second fitness function, that is

$$Appro(C(x)) = \left| f_p'(x) \otimes x \oslash f_p(x) \right| \tag{9}$$

We use the *Fit2* function to guide the search algorithm to find the possible inputs that may trigger the ill-conditioned problem (if there exists) in a numerical program. The approximation ($C(x)$ to $Appro(C(x))$) makes us free from the exact calculation of the mathematical function $f(x)$ which needs high precision and may lead to high time overhead.

Algorithm 1 The DEMC algorithm

```
Input: $f_p$, $f$, $I_{init}$, $[Fit1 : ErrBits(f, f_p)]$, $[Fit2 : Appro(C(x))]$
Output: $(MaxErr, x_m)$
 1: $tmp\_list \leftarrow [\,]$
 2: $error\_list \leftarrow [\,]$
 3: $L_I \leftarrow Partition(I_{init})$
 4: for $I_i \in L_I$ do
 5:     $x_i \leftarrow$ DIFFERENTIAL_EVOLUTION$(I_i, Fit2)$
 6:     $(err_i, x_i) \leftarrow Fit1(x_i)$
 7:     $tmp\_list.append((err_i, x_i))$
 8: end for
 9: $tmp\_list \leftarrow SortByError(tmp\_list)$
10: for $(err_i, x_i) \in tmp\_list$ do
11:     $(max\_err_t, x_t) \leftarrow$ MCMC$(x_i, Fit1)$
12:     $error\_list.append((max\_err_t, x_t))$
13: end for
14: $(MaxErr, x_m) \leftarrow SortByError(error\_list)[0]$
15: return $(MaxErr, x_m)$
```

The DEMC algorithm is shown in Algorithm 1. The inputs of the DEMC algorithm include the numerical program f_p together with its corresponding mathematical function f, the input domain I_{init} of f_p, and two fitness functions (*Fit1* and *Fit2*). In brief, we first partition the search space (the whole input domain) into many smaller parts, then apply the differential evolution algorithm using *Fit2* (which can be calculated fast) as guide to search in each part, and finally call MCMC using *Fit1* as guide to refine the search results. As shown in Algorithm 1, the input domain is first partitioned into many smaller parts (subintervals) by the *Partition* function which is designed according to the distribution of floating-point numbers. After that, we first call the differential evolution algorithm to search for the input that can trigger large condition number in each small input subinterval I_i. Next, we perform the *MCMC* algorithm to find the possible maximum floating-point error max_err_t around each input x_i that is found previously by the differential evolution algorithm. In other words, we employ the MCMC algorithm to search higher errors in the neighborhood of x_i. Finally, the DEMC algorithm returns the maximum floating-point *MaxErr* and the corresponding input x_m.

Generating the target input interval (I_{err}). To generate the input interval I_{err} enclosing the neighbors of x_m with respect to the given error threshold ε, we propose a so-called Point-to-Bound (PTB) algorithm.

Intuitively, around the input x_m producing the maximum floating-point error, there may exist some other inputs causing high floating-point errors. According to the formula $Forward\_error \approx B \cdot C(x)$ (Eq. 7), a high floating-point error evoked by an ill-conditioned problem may decline with the decreasing of the value of condition number $C(x)$. Moreover, if the mathematical function is close to a linear function, the derivate $|f'(x)|$ of the mathematical function is almost a stable value in a small interval around x_m, and thus the value of $|f'(x) \cdot x|$ is also almost a stable value in a small interval around x_m, which means that the value of $|f(x)|$ becomes the main influential factor of condition number $C(x) = |f'(x) \cdot x/f(x)|$ (Eq. 4).

Based on above analysis, we know that floating-point errors triggered by inputs around x_m will decrease with the increasing of the value of $|f(x)|$. Meanwhile, the formula (Eq. 2) we use to evaluate the floating-point error is also implicitly related with the *ulp* value of $|f(x)|$. Thus we have the intuition that the floating-point errors triggered by inputs around x_m may be a stepwise decline trend with the value of $ulp(f(x))$. For example, as shown in Fig. 4(a), x_0 is the input triggering maximum error and also the input (closest to the root of the function) making $ulp(f(x))$ smallest, and the values of errors have a stepwise decline trend around x_0.

Algorithm 2 The PTB Algorithm

```
Input: $f_p, f, x_m, \varepsilon, step$
Output: $I_{err}$ : [$Lower\_bound$, $Up\_bound$]
        /* search the upper bound on the right of $x_m$*/
 1: $Up\_bound \leftarrow$ IterationForBound($f_p$, $f_r$, $x_m$, $\varepsilon$, $step$, $1$)
        /* search the lower bound on the left of $x_m$*/
 2: $Lower\_bound \leftarrow$ IterationForBound($f_p$, $f_r$, $x_m$, $\varepsilon$, $step$, $-1$)
 3: return [$Lower\_bound$, $Up\_bound$]
 4: function IterationForBound($f_p$, $f_r$, $x_m$, $\varepsilon$, $step$, $sign$)
 5:     $temp\_max\_error \leftarrow Error\_evaluation(x_m, f_p, f_r)$
 6:     $temp\_bound \leftarrow x_m$
 7:     $step \leftarrow ulp(x_m) \otimes 1000$
 8:     while $temp\_max\_error > \varepsilon$ do
 9:         $temp\_bound \leftarrow temp\_bound \oplus step \otimes sign$
10:         $temp\_max\_error \leftarrow Max\_error\_find(temp\_bound, step)$
11:         $times \leftarrow max([(temp\_max\_error \oslash \varepsilon), 2.0])$
12:         $step \leftarrow times * step$
13:     end while
14:     $last\_step \leftarrow step$
15:     $temp\_bound \leftarrow IterationBack(\varepsilon, temp\_bound, f_p, f, last\_step)$
16:     return $temp\_bound$
17: end function
```

We design the PTB algorithm mainly based on the possible stepwise decline trend of floating-point errors triggered by inputs around x_m. The PTB algorithm is shown in Algorithm 2. The inputs of the PTB algorithm include the numerical program f_p together with its corresponding mathematical function f, the input x_m as the starting point, the error threshold ε, and an initial *step* for search bound. The output of the algorithm is the target I_{err}.

To generate an I_{err} as small as possible which includes all inputs around x_m that can trigger floating-point error higher than the given error threshold ε, we first search a tight upper bound (*Up_bound*) of I_{err} which is larger than x_m (Line 1). The temporary value of *Up_bound* is saved in variable *temp_bound* (Line 6) and variable *step* (Line 7) is used to refresh the value of *temp_bound*. We use variable *temp_max_error* to record the local maximum floating-point errors around *temp_bound*. Note that *Max_error_find* (Line 10) finds the maximal floating-point error in a small interval (decided by the value of step) around *temp_bound*. The value of $times$ is large than 2.0 and the value of $temp\_max\_error \oslash \varepsilon$ is used to adjust (Line 12) the value of $step$. *temp_bound* will keep changing by $step \otimes sign$ until $temp\_max\_error \leq \varepsilon$ (Lines 8 - 9). The value of *step* may increase too large, which leads *temp_bound* to be far away from the possible smallest *Up_bound*, so we call the IterationBack function to refine the *last_step* to find a tighter *Up_bound*. Then, we perform a similar process to find the possible largest lower bound *Lower_bound* of I_{err}. After finding the lower bound and upper bound of I_{err} (Lines 1 - 2), the algorithm will return the target input interval: $I_{err} = [Lower\_bound,\ Up\_bound]$.

4.2 Deriving an Approximation of Mathematical Function

In this section, we describe how to derive an approximation of a mathematical function within a small input interval I_{err} to satisfy a given error threshold ε. First, we give the following definition of an approximation of a mathematical function.

Definition 3. *Given an error threshold ε and an input interval I_{err}, for a mathematical function $f : \mathbb{R} \to \mathbb{R}$, if there exists a function $\overline{f} : \mathbb{F} \to \mathbb{F}$, such that for any floating-point input $x_i \in I_{err}$, $ErrBits(\overline{f}(x_i),\ f(x_i)) \le \varepsilon$ holds, then $\overline{f}$ is said to be an approximation of f in the given I_{err} that satisfies the given error threshold ε.*

Based on the above definition, we have the following theorem:

Theorem 4.1. *Given a mathematical function $f : \mathbb{R} \to \mathbb{R}$ and an input interval I_{err}, for any $\varepsilon \in [0, 64)$ and any floating-point input $x_i \in I_{err}$, there exists $\overline{f} : \mathbb{F} \to \mathbb{F}$ such that*

$$ErrBits(\overline{f}(x_i), f(x_i)) \le \varepsilon$$

Proof. Construct a dictionary $D_f = \{x_i : o_i | x_i \in I_{err}, o_i = fl(f(x_i))\}$, and let $\overline{f} = D_f$. Then $ErrBits(\overline{f}(x_i), f(x_i)) = ErrBits(fl(f(x_i)), f(x_i)) = 0 \le \varepsilon$ (because $FPNum(fl(f(x_i)), f(x_i)) = 1$ according to Eq. 2). □

Theorem 4.1 shows that there always exists an approximation that can satisfy a given error threshold ε. Actually, from the proof, we can construct the approximation D_f that satisfies $\varepsilon = 0$. Based on Theorem 4.1, we design an iterative refinement algorithm that will terminate with an approximation $\overline{f}$ satisfying a given error threshold ε, e.g., $\overline{f} = D_f$ in the worst case. Note that the algorithm is general for any given I_{err} and ε, which means that the algorithm not only can be adopted to repair high floating-point errors evoked by ill-condition problems but also can be used for repairing floating-point errors in general.

To detail the algorithm, we first introduce two main components of the algorithm: linear approximation and error compensation.

Linear approximation. Based on Hypothesis 1 that high floating-point errors should be triggered by inputs localized in some small input intervals, it is natural to leverage the linear approximation to simulate a mathematical (univariate) function f within a small input interval.

For a given input interval I_{err}, the linear approximation can be easily depicted by a line segment geometrically, as shown in Fig. 7. The two end points of the line segment can be determined by $fl(f(x))$. Let us consider, in Fig. 7, three points $(x_i,\ O_i)$, $(x_k,\ O_k)$, $(x_j,\ O_j)$ where $\{x_i,\ x_k,\ x_j\} \in I_{err}$, $O_i = fl(f(x_i))$, $O_k = fl(f(x_k))$, $O_j = fl(f(x_j))$. We draw the line l to approximate the three points, where we fix the two end points to satisfy $O_i{}^l = O_i = fl(f(x_i))$ and

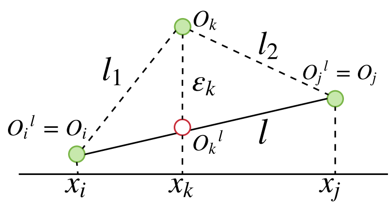


Fig. 7. Schematic diagram of linear approximation

${O_j}^l = O_j = fl(f(x_j))$. Then we have $ErrBits({O_i}^l,\ f(x_i)) = 0$, $ErrBits({O_j}^l,\ f(x_j)) = 0$, which means that at the two end points the linear approximation induces 0 bit error compared with the original mathematical function. Then, the intermediate points of the line segment can be calculated by a linear function. The linear function can be implemented as

$$f_l(x) = k \otimes (x \ominus x_s) \oplus y_s \tag{10}$$

where k is the slope of the line segment, $x \in I_{err}$, (x_s, y_s) is one end point of the line segment.

Finally, the linear approximation overall can be described as:

$$\textit{For any floating point input } x \in I_{err},\ l(x) = \begin{cases} fl(f(x)) & \textit{if } x \in \{x_s,\ x_e\} \\ \\ f_l(x) & \textit{if } x \notin \{x_s,\ x_e\} \end{cases} \tag{11}$$

where $\{x_s,\ x_e\}$ is the x-axis values of the two end points of the line segment.

Theorem 4.2. *Given a mathematical function $f : \mathbb{R} \to \mathbb{R}$ and an input interval I_{err}, for any $\varepsilon \in [0,\ 64)$ and any floating-point input $x_i \in I_{err}$, there exist a piecewise linear function $\overline{\Sigma}_l(x) : \mathbb{F} \to \mathbb{F}$, such that*

$$ErrBits(\overline{\Sigma}_l(x),\ f(x_i)) \le \varepsilon$$

Proof. Let the set $\{x_0,\ x_1, ...,\ x_n\}$ denote all floating-point numbers in I_{err}. We construct a piecewise linear function $\overline{\Sigma}_l(x) = l_k(x)$ when $x \in \{x_k, x_{k+1}\}$ $(k \le n-1, k \in \mathbb{N}\})$, where $l_k(x)$ represents the linear approximation over $I_{err} = [x_k, x_{k+1}]$, following the definition in Eq. 11. Then for any $x_i \in I_{err}$, $\overline{\Sigma}_l(x_i) = l_i(x_i)$. According to Eq. 11, we have $l_i(x_i) = fl(f(x_i))$, and thus we have $ErrBits(\overline{\Sigma}_l(x_i),\ f(x_i)) = ErrBits(fl(f(x_i)),\ f(x_i)) = 0 \le \varepsilon$. □

Theorem 4.2 shows that for a mathematical function (over floating-point inputs), there always exists a piecewise linear function that can satisfy a given error threshold ε, e.g., $\overline{\Sigma}_l(x)$ in the worst case. Note that $\overline{\Sigma}_l(x)$ is equivalent to D_f (in Theorem 4.1). Based on Theorem 4.2, we can continuously reduce the error at the point that does not satisfy the error threshold ε yet by creating a new line segment. After a limited iteration number, we can find a piecewise linear function $\Sigma_l(x)$ to satisfy $ErrBits(\Sigma_l(x_i), f(x_i)) \le \varepsilon$, and in the worst case we can have $\Sigma_l(x_i) = \overline{\Sigma}_l(x_i)$. For example, for the intermediate point $(x_k,\ O_k)$ in Fig. 7, a possible error ε_k exists between ${O_k}^l$ and O_k. If ε_k is smaller than a given error threshold ε, we say that we have already found a linear approximation for $f(x)$ at the three points, otherwise, a piecewise linear function $\Sigma_l(x)$ consisting of two new line segments (l_1 and l_2) will replace $l(x)$, to make $ErrBits(\Sigma_l(x_k), f(x_k)) = 0 \le \varepsilon$.

Error compensation Using purely linear approximation may result in too many pieces of linear functions to express an accurate enough approximation of a mathematical function even in a small input interval I_{err}. Therefore, we add an error compensation on each piece of the piecewise linear function to reduce the error between the linear approximation and the mathematical function.

To conduct the error compensation on each linear function $l(x)$, we first introduce a function $AbsErr(x) : \mathbb{F} \to \mathbb{F}$ to express the absolute error between a linear function and the original mathematical function:

$$\textit{For any } x \in I_{err},\ AbsErr(x) = fl(f(x)) \ominus l(x) \tag{12}$$

The calculation of the function $AbsErr(x)$ needs to calculate the mathematical function $f(x)$ (which cannot be implemented exactly in our patch using finite precision), so we use a error compensation function to approximate $AbsErr(x)$. According to Eq. 11, for the two end points of a line segment, we have $AbsErr(x_s) = AbsErr(x_e) = 0$. Consider the possible nonlinear feature of the mathematical function in I_{err} and the fact that we already know two roots $(x_s,\ x_e)$ of a quadratic function, we use the following quadratic function to approximate $AbsErr(x)$:

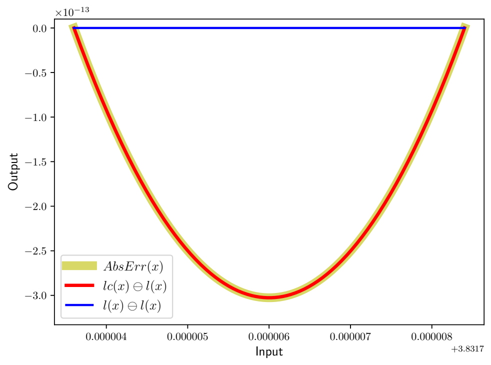
$$For\ any\ x \in I_{err},\ AbsErr(x) \approx \lambda \otimes (x \ominus x_s) \otimes (x \ominus x_e) \tag{13}$$

We choose the middle point x_m of x_s and x_e to calculate the value of λ. Then we have $\lambda = AbsErr(x_m) \oslash ((x_m \ominus x_s) \otimes (x_m \ominus x_e))$ wherein $AbsErr(x_m)$ can be computed though $fl(f(x_m)) \ominus l(x_m)$ following Eq. 12.

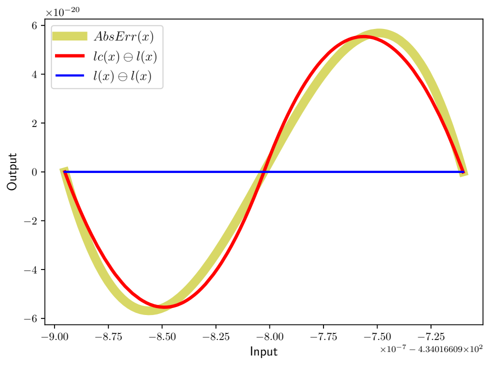
Finally, we define the error compensation function as $c(x) = \lambda \otimes (x \ominus x_s) \otimes (x \ominus x_e)$ and add it back to the linear function. Then we can have the following possible more accurate approximation of a mathematical function:

$$For\ any\ x \in I_{err},\ lc(x) = l(x) \oplus c(x) \tag{14}$$

The error compensation helps us to get a more accurate approximation of a mathematical function. For example, as shown in Fig. 8(a), for the mathematical function *bessel_J1*, we compare $AbsErr(x)$ with $lc(x) \ominus l(x)$ to show the effectiveness of error compensation, where we use the $l(x) \ominus l(x)$ as the baseline. From Fig. 8(a), we can see that for this example one line segment with one time of error compensation (red line) fits well the original mathematical function (bold yellow line).



(a) Error compensation on *bessel_J1*



(b) Error compensation on *airy_Ai*

Fig. 8. Examples of error compensation

However, not all functions can be well approximated to satisfy a given error threshold ε by a line segment with one time of error compensation. E.g., as shown in Fig. 8(b), even though we have two line segments with error compensations, it may still be not accurate enough for a small error threshold ε and thus more line segments with error compensations may be needed to satisfy the ε. Thus, we propose an iterative refinement algorithm to call the previous two steps (linear approximation and error compensation) iteratively to satisfy a given error threshold ε.

Iterative refinement The iterative refinement algorithm iteratively applies linear approximation and error compensation to generate a piecewise quadratic function $\overline{f}$ that can satisfy the given error threshold ε.

As shown in Algorithm 3, the inputs of the iterative refinement algorithm include the mathematical function f, the given error threshold ε and input interval I_{err} : $[Lower\_bound,\ Up\_bound]$ that is the output of Algorithm 2. In Algorithm 3, we maintain a global list *Line_list* to store all pieces of linear functions with error compensations and return it as the output of the algorithm.

Algorithm 3 first calls the function *LinearApproximate* (Line 6) to generate a linear function l. After getting l, the algorithm calls the function *ErrorCompensation* (Line 7) to generate the error compensation c for l. After getting l and c, the algorithm generates the approximation function $\overline{f}$ (Line 8) for f within I_{err}. Then, the maximum error $MaxErr_{\overline{f}}$ between $\overline{f}$ and f is found by the function *MaxErrorSearch* (Line 9). The recursive function IterLineApprox will exit if the value of $MaxErr_{\overline{f}}$ is smaller than the given threshold ε and at the same time the linear approximation with error compensation (i.e., (l, c)) will be stored in the global list *Line_list* (Line 11). Otherwise, two new input intervals $[Lower\_bound,\ x_{next}]$ and $[x_{next},\ Up\_bound]$ will be passed to the recursive function to do another new iteration on the new input intervals, where x_{next} is the found input that triggers the possible maximum error $MaxErr_l$ in l.

Algorithm 3 Iterative Refinement Algorithm

```
Input: $f,\ \varepsilon,\ I_{err} :\ [Lower\_bound,\ Up\_bound]$
Output: $Line\_list$
 1: $Line\_list \leftarrow [\ ]$
 2: IterLineApprox($f,\ \varepsilon,\ [Lower\_bound,\ Up\_bound]$)
 3: return $Line\_list$
 4: function IterLineApprox($f,\ \varepsilon,\ [Lower\_bound,\ Up\_bound]$)
 5:     global $Line\_list$
 6:     $l \leftarrow LinearApproximate([Lower\_bound,\ Up\_bound],\ f)$
 7:     $c \leftarrow ErrorCompensation([Lower\_bound,\ Up\_bound],\ f,\ l)$
 8:     $\overline{f} \leftarrow l \oplus c$
 9:     $(MaxErr_{\overline{f}},\ x_{\overline{f}}) \leftarrow MaxErrorSearch(f,\ \overline{f}, [Lower\_bound,\ Up\_bound])$
10:     if $MaxErr_{\overline{f}} \leqslant \varepsilon$ then
11:         $Line\_list.append(\ (l,\ c))$
12:         return 0
13:     else
14:         $(MaxErr_l,\ x_{next}) \leftarrow MaxErrorSearch(l,\ f)$
15:         IterLineApprox($f,\ \varepsilon,\ [Lower\_bound,\ x_{next}]$)
16:         IterLineApprox($f,\ \varepsilon,\ [x_{next},\ Up\_bound]$)
17:     end if
18: end function
```

Termination guarantee. The termination of iterative refinement algorithm can be proved based on the finiteness of floating-point numbers and Eq. 11. For each iteration, if the $MaxErr_{\overline{f}} > \varepsilon$, a new input x_{next} will be chosen as the x value of a new end point which is then used to generate two new line segments. According to Eq. 11, we have $ErrBits(\overline{f}(x_{next}),\ f(x_{next})) = 0$, which means that at least the floating-point error triggered by the input x_{next} is reduced to zero after one iteration. Therefore, if there exists n ($n \geq 2$ and $n \in \mathbb{N}$) floating-point numbers in the initial input interval I_{err}, in the worst case, after $2 * n - 3$ iterations (which can be easily proved by mathematical induction), i.e., when all inputs are included as the end points of lines, the final resulting approximation of f will be the function $\overline{\Sigma}_l(x)$ (in Theorem 4.2) and then the algorithm will terminate (since $MaxErr_{\overline{f}}$ becomes 0 for all floating-point inputs in I_{err}). The iterative refinement algorithm guarantees that the algorithm can always find an approximation for a mathematical function f on the input interval I_{err} to satisfy a given error threshold ε in finite steps. Note that the termination of iterative refinement algorithm is not influenced by the mathematical feature of the mathematical function f.

Effectiveness analysis. The effectiveness of the algorithm is mainly influenced by the mathematical feature of the mathematical function f. For example, as shown in Fig. 8(a), the *bessel_J1* is not a quadratic function, but it has a quadratic polynomial feature in a small input interval and thus can be well approximated by the $lc(x)$ function (in Eq. 14), while the function *airy_Ai* (in Fig. 8(b)) may need more iterations to satisfy the same given error threshold ε.

4.3 Producing Patch

After deriving an approximation of the target mathematical function f, we illustrate how to convert the approximation to a patch in practice.

First, accurately implementing approximations is critical for approximating mathematical functions. To improve the accuracy of implementation of the linear function (Eq. 10) and the error compensation function (Eq. 13), we propose a strategy from the implementation aspect to make the subtraction operations exact (in Eq. 10 and Eq. 13).

In detail, we partition the target input interval I_{err} into a set of smaller input intervals $D(I_{err})$ according to the distribution of floating-point numbers, such that for each $I_s \in D(I_{err})$, we have $ulp(x_i) = ulp(x_j)$ and $x_i \otimes x_j \geq 0$ for any $(x_i, x_j) \in I_s$. Let the set $X_s = \{x_0, x_1, ..., x_n\}$ represent all floating-point numbers in I_s in an increasing order. When both $x_i, x_j \in X_s$ are denormalized floating-point numbers, $x_i - x_j$ can be exactly represented by denormalized floating-point representation and $x_i \ominus x_j = x_i - x_j$.

Now, we consider the case that both x_i and x_j are normalized floating-point numbers. We leverage the Sterbenz's theorem [Sterbenz 1973]:

Theorem 4.3. *Let $x, y \in \mathbb{F}$ and $x, y \geq 0$. Then*

$$\frac{x}{2} \leq y \leq 2x \Longrightarrow x \ominus y = x - y$$

Theorem 4.3 can be easily extended to cover the case $x, y \leq 0$, as the following one:

Theorem 4.4. *Let $x, y \in \mathbb{F}$. Then*

$$(\frac{x}{2} \leq y \leq 2x) \vee (2x \leq y \leq \frac{x}{2}) \Longrightarrow x \ominus y = x - y$$

For any two floating-point inputs $x_i, x_j \in X_s$, we know $ulp(x_i) = ulp(x_j)$ and $x_i \otimes x_j \geq 0$. When $x_i, x_j \geq 0$, if $x_i \in [2^k, 2^{k+1})$, then $x_j \in [2^k, 2^{k+1})$, so we have $(\frac{x_i}{2} \leq x_j \leq 2x_i)$. When $x_i, x_j \leq 0$, if $x_i \in (-2^{k+1}, -2^k]$, then $x_j \in (-2^{k+1}, -2^k]$, so we have $(2x_i \leq x_j \leq \frac{x_i}{2})$. Then according to Theorem 4.4, we have $x_i \ominus x_j = x_i - x_j$.

Overall, we have:

$$\textit{For any two floating point inputs } x_i, x_j \in X_s, \; x_i \ominus x_j = x_i - x_j \tag{15}$$

Eq. 15 ensures that the $\ominus$ operations in the linear approximation function (Eq. 10) and the error compensation function (Eq. 13) are exact on each $I_s \in D(I_{err})$.

Then, we store each piece of the approximation $\overline{f_i}$ that is generated for each $I_i \in D(I_{err})$ using a 6-tuple structure. As shown in Algorithm 3 (Line 11), $\overline{f_i}$ is expressed by a list of (l, c). We use the following 6-tuple to store each (l, c) in detail

$$st_j^{(i)} = \langle k_j^{(i)}, x_{s_j}^{(i)}, y_{s_j}^{(i)}, x_{e_j}^{(i)}, y_{e_j}^{(i)}, \lambda_j^{(i)} \rangle \tag{16}$$

where $k_j^{(i)}$ is the slope of linear function (Eq. 10) , $(x_{s_j}^{(i)}, y_{s_j}^{(i)})$ is the starting point of l_j, $(x_{e_j}^{(i)}, y_{e_j}^{(i)})$ is the end point of l_j, $\lambda_j^{(i)}$ is the coefficient of the error compensation function (Eq. 13). Let m represent the length of the list of (l, c) and the piecewise quadratic function $\overline{f_i}$ on I_i be stored as

$ST^{(i)} = \{st_0^{(i)},\ st_1^{(i)},\ \ldots,\ st_m^{(i)}\}$ where $st_k^{(i)}$ $(0 \le k \le m)$ stores the k-th piece of the $\overline{f_i}$ and for any two $st_j^{(i)},\ st_{j+1}^{(i)} \in ST^{(i)}$ $(0 \le j \le m-1)$ we have $x_{e_j}^{(i)} = x_{s_{j+1}}^{(i)}$.

Next, we derive a search optimization problem of finding the right $st_r^{(i)} \in ST^{(i)}$ to calculate $\overline{f_i}(x)$ for each floating-point input $x \in I_i$.

Search optimization To do the search optimization, we first extract the x-values of the end points from $ST^{(i)}$ and store them as a list $L_e = [0, x_{e_0}^{(i)}, x_{e_1}^{(i)}, \ldots, x_{e_m}^{(i)}]$. Given an input $x_t \in I_i$, if $L_e[j] \le x_t \le L_e[j+1]$, we will use $st_j^{(i)}$ to calculate $\overline{f_i}(x_t)$. For the input $x_t \in I_i$, it is a search problem to find the right j such that $L_e[j] \le x_t \le L_e[j+1]$. We transfer the search problem into a function $f_\Delta : X_i \to \{0, \ldots, m\}$ where X_i represents all floating-point numbers in I_i in increasing order, and for each $x_t \in I_i$, when $L_e[j] \le x_t \le L_e[j+1]$, we have $f_\Delta(x_t) = j$. Furthermore, when the value of m is large, we use a polynomial function $P_i(x)$ to fit the inputs and outputs of f_Δ such that $f_\Delta(x_t) \approx j_e$ where $j_e = floor(P_i(x_t))$.

More specifically, we use the least squares method to generate a polynomial function (i.e. $P(x) = p[0] * x^n + \ldots + p[n]$ where $p[i]$ $(i \in [0, \ldots, n])$ is the coefficient we need to solve according to a given data set) from the given data set such that inputs are $[x_{e_0}^{(i)}, x_{e_1}^{(i)}, \ldots, x_{e_m}^{(i)}]$ and their corresponding outputs are $[0, \ldots, m]$. Then we store the polynomial function in the patch and call it when needed to estimate the j as shown in Line 3 of Fig. 9. When the value of m is small we employ a binary search to find j.

```
1: procedure PATCH_CODE($x$)
2:     Array: $ST^{(i)}$, $L_e$
3:     $j_e \leftarrow floor(P(x))$
4:     $j \leftarrow Search\_idx(j_e,\ L_e,\ x)$
5:     $(l,\ c) \leftarrow transfer(ST^{(i)}[j])$
6:     $\overline{f} \leftarrow l \oplus c$
7:     return $\overline{f}(x)$
8: end procedure
```

Fig. 9. Pseudocode of patch

Finally, for a floating-point input $x \in I_i$, the patch first calls a polynomial function P_i to estimate the possible j value, then searches for the exact j and calls $st_j^{(i)}$ to generate $\overline{f}$ for calculating $\overline{f}(x)$. The pseudocode in Fig. 9 shows the process.

5 IMPLEMENTATION AND EVALUATION

5.1 Implementation

In this section, we introduce AutoRNP, the prototype tool that implements our approach. The implementation follows the work flow in Fig. 3, consisting of the following components:

(S1) *A detector of high floating-point errors* which, given an input domain I_{init}, numerical program f_p and the corresponding mathematical function f, finds the I_{err} of f_p. We use the package *mpmath* [Johansson et al. 2013] which is an open-source Python library for real and complex floating-point arithmetic with arbitrary precision. As explained in Sect. 4.1, the detector involves the uses of differential evolution and MCMC algorithms to implement the DEMC algorithm. We use differential evolution algorithm [Storn and Price 1997] and the Basinhopping algorithm [Wales and Doye 1997] as the MCMC engine. Both of them are available from Scipy (version 1.0.0)[5].

(S2) *An extractor* that extracts an approximation $\overline{f}$ of the mathematical function f in I_{err} to satisfy a given error threshold ε.

(S3) *A patch generator* that applies the search optimization on the approximation $\overline{f}$ and converts $\overline{f}$ to patch. Our tool is designed for numerical programs in GSL, so the patch is stored in C code. We use the *polyfit* function in numpy[6] to produce the polynomial function P for the search optimization and evaluate the polynomial function P by Horner's method [Cajori 1911].

[5]https://docs.scipy.org/doc/scipy-1.0.0/reference/optimize.html
[6]https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html

5.2 Benchmarks and Experimental Setup

Our subjects are chosen from the GNU Scientific Library[7] (GSL) (version 2.1). GSL has in total 154 functions with all inputs being floating-point numbers and the output being one floating-point number. Note that 67% of those functions (104 of 154) are univariate functions, and many multi-argument functions are built based on those univariate functions. Our prototype tool currently only supports repairing numerical programs with one input. To achieve the accurate outputs of mathematical functions that those programs implemented, we choose *mpmath* to supply implementations of those mathematical functions under arbitrary precision to avoid unexpected errors that may be introduced by higher precision execution on original programs (see details in §6). So we have to remove those univariate functions that are not supported in *mpmath* and we get 49 numerical programs as our initial subjects. Then we use the DEMC algorithm (§4.1) and run it 100 times on each program to find the maximum floating-point error of those 49 numerical programs and finally choose 20 numerical programs which have significant higher maximum floating-point errors (measured by *ErrBits*) than other 29 programs (whose *MaxErr*'s are less than 30). The benchmark is shown in Table 1. We also notice that there may exist not only one I_{err} in a numerical program, and we choose to repair the I_{err} around the input that can trigger the possible maximum floating-point error of a numerical program over the whole input domain. Note that the repair process for each I_{err} is independent and repeatable.

Table 1. Benchmark from GSL

ID	Program	*MaxErr*	ID	Program	*MaxErr*
P1	gsl_sf_airy_Ai	62.94	P11	gsl_sf_legendre_P2	49.75
P2	gsl_sf_airy_Bi	62.94	P12	gsl_sf_legendre_P3	61.92
P3	gsl_sf_airy_Ai_deriv	62.94	P13	gsl_sf_legendre_Q1	52.01
P4	gsl_sf_airy_Bi_deriv	62.94	P14	gsl_sf_psi	62.94
P5	gsl_sf_bessel_J0	53.03	P15	gsl_sf_Chi	50.47
P6	gsl_sf_bessel_J1	61.92	P16	gsl_sf_Ci	62.92
P7	gsl_sf_bessel_Y0	61.92	P17	gsl_sf_lnsinh	61.92
P8	gsl_sf_bessel_Y1	51.31	P18	gsl_sf_zeta	51.35
P9	gsl_sf_clausen	61.92	P19	gsl_sf_eta	51.44
P10	gsl_sf_expint_Ei	51.76	P20	gsl_sf_psi_1	52.57

Performance on real-world numerical programs We want to evaluate the performance of our approach on real-world numerical programs (see details within §5.3). First, we investigate the repair ability of AutoRNP under a given criterion of repair and a time limit. We use AutoRNP to repair high floating-point errors in the 20 numerical programs of GSL. We set three levels of error thresholds. The error threshold is calculated by the following formulas:

$$MeanErr(f_p) = \sum_{i=0}^{k} \frac{ErrBits(f(x_i), f_p(x_i))}{k} \tag{17}$$

$$\varepsilon = (MaxErr(f, f_p) - MeanErr(f_p)) * \tau + MeanErr(f_p) \tag{18}$$

where $MeanErr(f_p)$ calculates the mean error of f_p over its whole input domain, and k is the number (more than 10 million) of sampling inputs. We set $\tau = \{H : 0.1;\ M : 0.2;\ L : 0.3\}$ to calculate high H_ε, middle M_ε and low L_ε level threshold for repair. Note that if $\tau = 0$, the threshold ε equals to the mean error $MeanErr(f_p)$. We set time limit as 3 hours for one time of repairing a numerical program.

[7]http://www.gnu.org/software/gsl/

To evaluate the repair results, we define the accuracy of repair using the following formula:

$$AccRepair = \frac{PassNum}{TestNum} * 100\% \tag{19}$$

where $PassNum = |\{x|ErrBits(f(x), f_p(x)) \leq \varepsilon,\ x \in I_{err}\}|$, $TestNum = Min(|\{x|x \in I_{err}\}|, 100000)$. As shown in Eq. 19, we evaluate the accuracy of repair by checking how many inputs whose floating point errors of the outputs of $f_p(x)$ are less than the given error threshold ε in *TestNum* times random sampling tests.

Moreover, we also investigate the change of the maximum floating-point error and the average floating-point error over the input interval I_{err} after repairing.

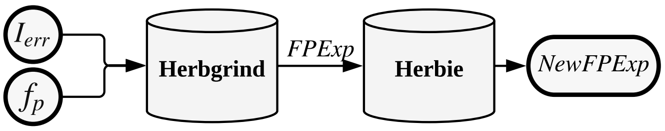


Fig. 10. Repair processes of HBG.

Comparing with state-of-the-art tools To compare with state-of-the-art methods, we build a connection between two state-of-the-art tools Herbgrind [Sanchez-Stern et al. 2018] and Herbie [Panchekha et al. 2015] to construct a new tool HBG. The work flow of HBG is shown in Fig. 10. In brief, HBG employs Herbgrind to find the root cause of floating-point errors during the execution of numerical program f_p on the given test inputs and extract floating-point expressions *FPExp* that can be recognized by Herbie. Then, HBG employs Herbie to use mathematical rewriting rules over *FPExp* to automatically improve the accuracy of the floating-point expressions. Finally, Herbie will output new floating-point expressions *NewFPExp*. Because Herbgrind does not include a functionality to detect high floating-point errors, we use our tool to generate test inputs in I_{err} for Herbgrind. For each I_{err} under different error thresholds, we generate 100 000 random sample points to aid Herbgrind to produce *FPExp*.

Influence of patches on the original programs Concerning the influence of patches on subjects, we mainly focus on the time overhead of the subjects after repairing, and we also consider the storage overhead of patches. Moreover, we discuss the readability of subjects after repairing. All our experiments were conducted on Ubuntu 16.04.3 LTS with 3.5GHz(8C) Intel Core i7-3770K CPU and 16 GB RAM.

5.3 Performance on Real-World Numerical Programs

Table 2 shows the repair time and the accuracy of repair. We compare the value of *AccRepair* (in Eq. 19) before repair (in "Before" column) and after repair (in "After" column) in the column "Accuracy of Repair".

As shown in Table 2, AutoRNP completes the repairing of 19 numerical programs in GSL, while accounts time out (TO) for *P20*. For *P20*, we further investigate the reason of time out, and we find that the reason is mainly due to the mathematical feature of the function *gsl_sf_psi_1* (i.e., *P20*). The *gsl_sf_psi_1* in its I_{err} appears as an exponential function, while our approach relies on the linear approximation and the compensation of a quadratic function, which is not good enough to approximate an exponential function. In fact, we have rerun AutoRNP on *P20* without given a time limit

```
int gsl_sf_psi_1(const double x)
{ ...
  else if(x > -5.0)
  {/*Abramowitz + Stegun 6.4.6*/
    ...}
  else
  {/*Abramowitz + Stegun 6.4.7*/
    ...}
}
```

Fig. 11. Code of *gsl_sf_psi_1*

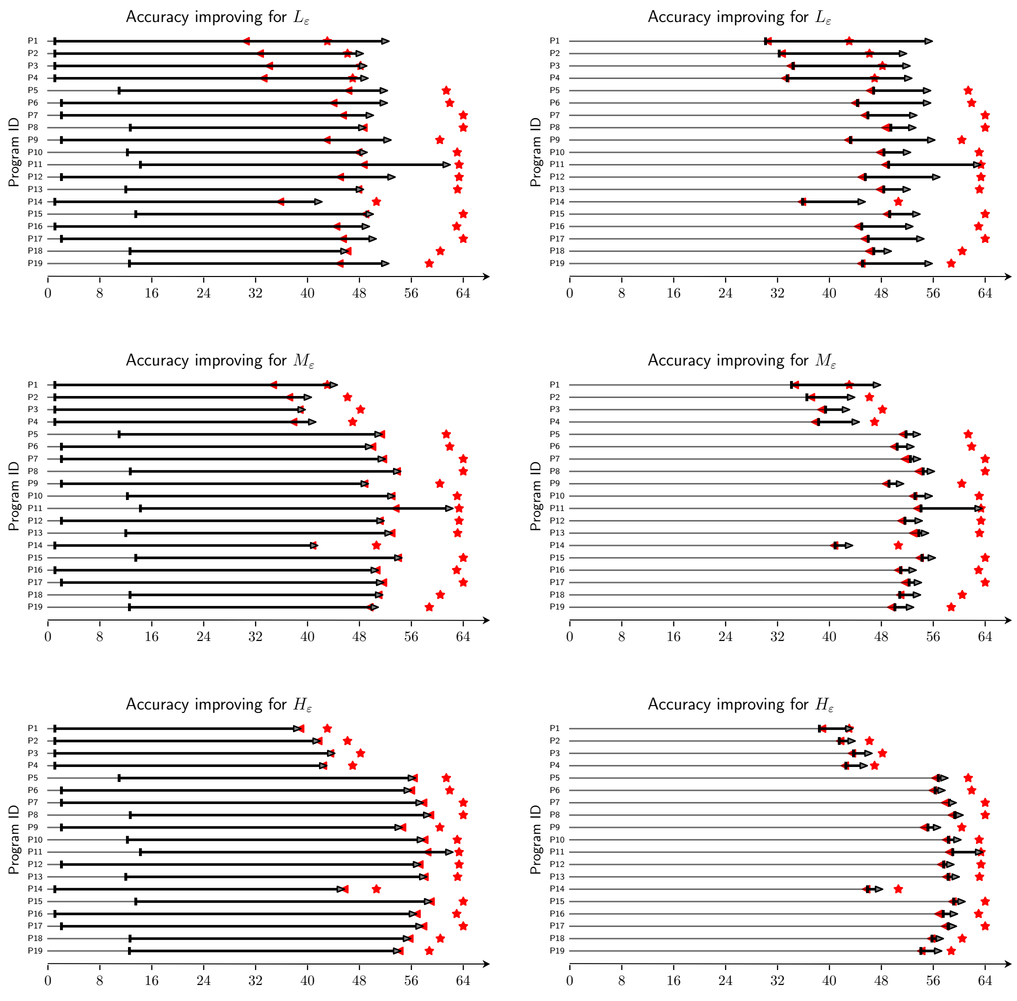
Table 2. Experimental results on 20 numerical programs in GSL

ID	Repair Time(s)			Accuracy of Repair					
	L_ε	M_ε	H_ε	L_ε		M_ε		H_ε	
				Before	After	Before	After	Before	After
P1	17.26	23.69	26.57	54.08%	100.00%	46.61%	100.00%	46.60%	100.00%
P2	22.51	21.67	35.25	51.16%	100.00%	40.52%	100.00%	55.50%	100.00%
P3	21.36	25.07	44.31	59.62%	100.00%	65.81%	100.00%	55.26%	100.00%
P4	21.73	22.40	34.75	60.98%	100.00%	65.85%	100.00%	52.94%	100.00%
P5	2.46	4.59	34.04	62.88%	100.00%	61.49%	100.00%	60.44%	100.00%
P6	2.04	2.33	36.02	58.57%	100.00%	61.63%	100.00%	59.12%	100.00%
P7	10.92	23.03	822.38	62.28%	100.00%	69.14%	100.00%	67.99%	100.00%
P8	59.95	150.73	4919.04	70.25%	100.00%	67.01%	100.00%	63.15%	100.00%
P9	66.57	89.18	1040.95	57.64%	100.00%	61.04%	100.00%	65.42%	100.00%
P10	1.90	4.93	95.70	66.70%	100.00%	58.40%	100.00%	61.92%	100.00%
P11	2.77	3.45	11.93	62.20%	100.00%	64.54%	100.00%	66.12%	100.00%
P12	2.32	4.38	63.30	63.75%	100.00%	62.91%	100.00%	59.64%	100.00%
P13	32.07	96.11	3561.67	64.37%	100.00%	69.99%	100.00%	57.75%	100.00%
P14	1.67	2.95	20.74	50.05%	100.00%	51.49%	100.00%	52.53%	100.00%
P15	3.51	9.62	411.72	60.20%	100.00%	58.63%	100.00%	56.79%	100.00%
P16	2.35	4.75	229.62	63.61%	100.00%	58.49%	100.00%	67.61%	100.00%
P17	2.01	5.25	124.59	63.35%	100.00%	66.38%	100.00%	63.55%	100.00%
P18	5.66	9.15	83.61	66.80%	100.00%	45.50%	100.00%	49.81%	100.00%
P19	10.62	17.15	52.30	51.77%	100.00%	61.94%	100.00%	45.93%	100.00%
P20	TO	TO	TO	54.97%	54.97%	52.53%	52.53%	59.23%	59.23%

and found that AutoRNP can complete the repair of *P20* to satisfy the low error threshold L_ε within 8 hours. Moreover, we find that GSL uses different formulas to implement the *gsl_sf_psi_1* function. As shown in Fig. 11, when the input x satisfying $x \leq -5$, the GSL program implements a formula (*6.4.7* in [Abramowitz 1974]) which is different with the case $x > -5.0$, while the mathematical function (in *mpmath*) keeps using the same formula (*6.4.6* in [Abramowitz 1974]) for both cases. If we change the implementation of *gsl_sf_psi_1* according to the formula *6.4.6* for $x \leq -5$, the high floating-point errors of *gsl_sf_psi_1* in I_{err} disappear.

What stands out in Table 2 is the "Accuracy of Repair" which is 100% for all three levels of error thresholds and for all programs except *P20*. The results show that most subjects have a polynomial feature in their small input interval I_{err} and can be well approximated by our approach.

Fig. 12 shows the bits correct for maximum error and average error after repair under different levels of error thresholds. Accuracy in Fig. 12(a) is measured by *Errbits*, and the maximum error



(a) Bits correct for maximum error (longer is better)

(b) Bits correct for average error (longer is better)

Fig. 12. Each row represents the improvement in accuracy achieved by AutoNRP on a single benchmark. The thick arrow starts at the accuracy of the program before repair, and ends at the accuracy of the program after repair. A **triangle** is drawn at the value of error threshold for each subject . A **pentagram** is drawn at the value of mean error of each subject in its whole input domain.

is found by our DEMC algorithm (in §4). Accuracy in Fig. 12(b) is measured by *Errbits*, averaged across 100 000 random input points in the I_{err} of a program[8]. As shown in Fig. 12, for all given thresholds (H_ε, M_ε, L_ε) and subjects excepts *P*20, AutoRNP successfully improves the accuracy such that the maximum error does not exceed the given error threshold and the resulting average error decreases. Interestingly, for *P11* , after repairing, its maximum error is almost getting close to the mean error as shown in Fig. 12. And the timings for repairing *P11* are also quite small, actually

[8]If the number of floating point inputs in I_{err} is less 100 000, then we test all floating-point inputs in I_{err}.

the least one for H_ε, as shown in Table 2. In fact, the *gsl_sf_legendre_P2* (i.e., *P11*) is a quadratic function which is a natural fit for our approach.

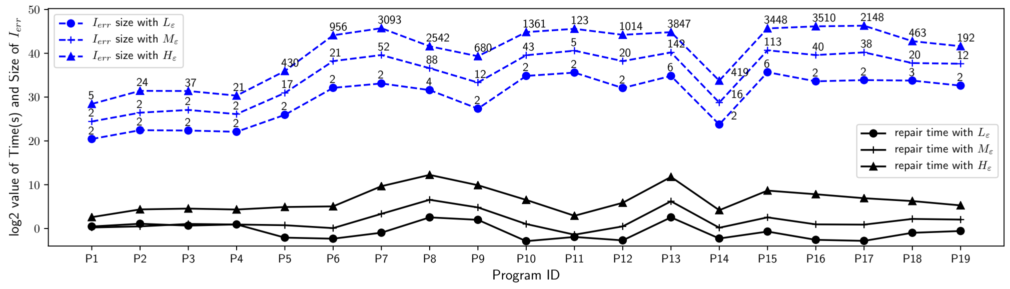


Fig. 13. Repair time and size of I_{err} under three level error thresholds

Fig. 13 shows the log_2 value of repair time and size of I_{err} for each subject under three thresholds. The size of I_{err} is the quantity of floating-point numbers in I_{err} (evaluated by Eq. 2). Note that we label the number of pieces in the approximation function $\overline{f}$ (i.e., the piecewise quadratic function) on each dot of the three lines of I_{err} (the blue lines in the upper of Fig. 13). As shown in Fig. 13, the higher level of error thresholds, the more repair time needed by our approach, the larger size of I_{err}, and the more pieces in the piecewise quadratic function.

In summary, the above results suggest that our approach can efficiently repair high floating-point errors in numerical libraries to satisfy a given error threshold.

5.4 Comparing with State-of-the-art Tools

We compare our experimental results with that given by HBG which integrates the two state-of-the-art tools Herbgrind and Herbie (as explained in §5.2). Because the main target of HBG is to reduce the average error rather that reducing only high errors over the input domain, we focus on the comparison of our AutoRNP with HBG over average bits correct over the same input domain I_{err}. In our experiments, we regard HBG as a black box and only consider the results reported by HBG.

As shown in Table 3, compared with HBG, AutoRNP keeps a more stable and higher improving of average bits correct for all 19 successful repaired programs (*P1* ~ *P19*) under the three levels of error thresholds. HBG fails to complete the repair process for *P1* and *P2* ("—" in Table 3), because Herbgrind does not produce any floating-point expressions that induce large floating-point errors and need to be repaired. This may be due to the fact that higher precision executions in Herbgrind do not find much differences in accuracy with the original precision executions for *P1* and *P2*. However, our AutoRNP successfully finds high floating-point errors for *P1* and *P2*, because AutoRNP compares the results of original program with that given by *mpmath* (which is consider as the mathematical function). Note that our approach aims to reduce higher errors rather than average errors but still results in lower average errors than HBG.

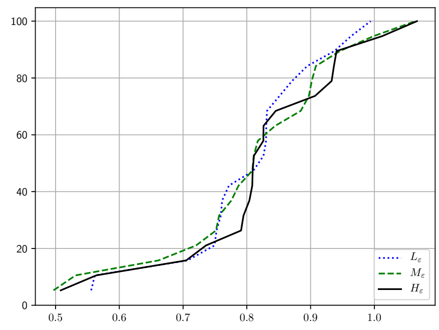
Overall, the experimental results in Table 3 show that HBG which uses mathematical rewriting technique might not fit for reducing high floating-point errors evoked by ill-conditioned problems (that are in nature of most of the subjects in §5.2). However, HBG (Herbgrind and Herbie) still is a very good combination to help developers to find implementations including high floating-point errors in numerical code and to supply suggestions for improving accuracy.

Table 3. AutoRNP vs HBG over average bits correct

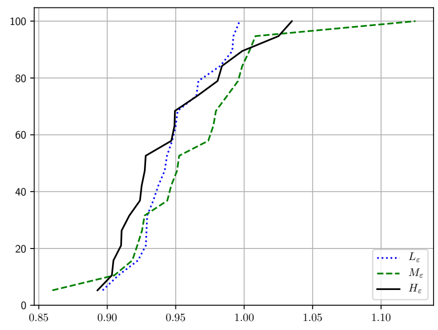
ID	Average error (before repair)			Improving of average bits correct (after repair, larger is better)					
	L_ε	M_ε	H_ε	L_ε		M_ε		H_ε	
				AutoRNP	HBG	AutoRNP	HBG	AutoRNP	HBG
P1	33.8	29.9	25.5	25.7	—	13.7	—	5.2	—
P2	31.7	27.5	22.5	19.6	—	7.4	—	2.5	—
P3	29.6	24.6	20.3	18.0	13.2	3.7	0.2	2.9	0.7
P4	30.5	25.7	21.5	19.2	6.3	6.3	0.2	3.3	0.0
P5	17.2	12.2	7.2	8.8	2.6	2.2	-0.1	1.5	0.0
P6	19.7	13.6	7.7	11.3	0.0	2.6	0.0	1.5	0.0
P7	18.1	11.5	5.5	7.6	0.1	1.6	0.0	1.0	0.1
P8	14.5	9.6	4.7	3.9	0.0	1.7	0.0	1.3	0.0
P9	20.8	14.8	8.8	13.0	0.5	2.3	TO	2.0	0.2
P10	15.6	10.8	5.7	4.1	0.0	2.7	0.0	1.9	0.0
P11	14.9	9.9	5.0	14.2	1.2	9.4	1.5	4.6	1.6
P12	18.5	12.4	6.4	11.5	1.7	2.7	0.4	1.6	1.0
P13	15.6	10.2	5.7	4.1	0.5	1.5	0.7	1.7	0.1
P14	28.1	23.1	18.1	9.7	0.0	2.7	0.0	2.3	0.0
P15	16.3	11.3	4.9	4.7	0.0	2.1	0.6	1.8	0.3
P16	14.7	9.7	6.5	7.9	0.1	2.4	0.0	2.2	-0.1
P17	19.0	13.0	5.7	8.6	0.7	1.9	0.7	1.3	0.5
P18	18.0	11.7	8.2	2.7	0.0	3.2	0.0	1.8	0.0
P19	17.2	13.2	9.9	10.8	0.0	2.9	0.0	3.2	0.0
P20	18.9	13.9	5.9	0.0	0.0	0.0	0.0	0.0	0.0

5.5 Influence of Patches on the Original Programs

Time overhead We timed the original program and the program after repaired via AutoRNP by running on 10 million random inputs in the I_{err} and the whole input domain respectively. Fig. 14 is the cumulative distribution of the slowdown for subjects after repaired. The horizontal axis shows the ratio between the run-time of the program after and before repair. Fig. 14(a) shows that more than *90%* programs after repaired by AutoRNP are faster (approximately 0.5 to 1.0) over inputs in I_{err}. When testing in the whole input domain, Fig. 14(b) shows the ratio of time overhead is around 1.0 (approximately 0.85 to 1.15). Overall, these results suggest that our patch does not slow down the execution in the whole input domain, while may accelerate the execution of subjects over I_{err}.



(a) Time overhead on I_{err}



(b) Time overhead on the whole input domain

Fig. 14. AutoRNP time overhead ratio (left is better) on 19 successfully repaired programs

Storage overhead Extra storages are needed for storing the patch. We evaluate the storage overhead by the size of patch file. As shown in Table 4, the storage overhead is increasing with the level of error thresholds. Note that there exist significant difference between the subjects over the storage overhead for the high level error threshold H_ε.

Table 4. Storage overhead (KB) on 19 successfully repaired programs

Threshold	Program ID									
	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
L_ε	1.68	1.69	1.82	1.82	1.73	1.73	1.73	2.04	1.69	1.73
M_ε	1.69	1.69	1.82	1.82	4.05	4.67	9.61	15.15	3.25	8.21
H_ε	2.15	5.10	7.42	4.78	67.92	149.08	479.39	393.82	107.46	211.58
	P11	P12	P13	P14	P15	P16	P17	P18	P19	
L_ε	1.78	1.77	2.39	1.60	2.22	1.58	1.67	3.01	2.82	
M_ε	2.24	4.54	23.38	3.79	18.89	7.59	7.37	4.39	4.37	
H_ε	20.69	157.09	591.47	66.51	533.47	542.30	332.95	72.46	32.44	

Readability The process (in §4.2) of deriving the approximation of mathematical function only bases on the input and output of f, and does not involve the implementation of numerical program. The patch is also independent of the implementation of original subject. Thus we can package the main implementation of a patch in a function and store it in an individual file. In the source code of original program, we just need to add a new branch to decide whether to call the function in the patch, as shown in Fig. 5. Our approach based on simple linear approximation and error compensation also makes the patch code (in Fig. 9) easily understand.

5.6 Wider Applicability

Our approach that is based on the mathematical feature of a numerical program also fits for the numerical program which implements the same mathematical function in other numerical libraries (e.g., the SciPy). We find 16 mathematical functions which are implemented in the 20 numerical programs (Table 1) are also implemented by 16 numerical programs in SciPy. We also detect high floating-point errors in 14 of those 16 numerical programs in SciPy on the same I_{err} (as GSL). We convert our approximation to python code and successfully repair those 14 numerical programs in SciPy.

6 DISCUSSION

Execution of mathematical function The higher precision execution of original numerical programs may introduce extra errors for numerical programs in numerical libraries that involve some precision-specific operations [Wang et al. 2016], so we do not use the higher precision execution as the choice for obtaining mathematical result. In our experiments, we use the package *mpmath* to supply the corresponding mathematical functions of original numerical programs in GSL. *mpmath* [Johansson et al. 2013] is an open-source library for real and complex floating-point arithmetic with arbitrary precision, and many computer algebra systems (e.g., SageMath) use *mpmath* as the underlying library. Since the corresponding functions supplied by *mpmath* are via arbitrary precision, we assume that they should not include precision-specific operations and thus can supply the (nearly) mathematical results of numerical programs in GSL.

Accuracy of repair A reasonable explanation for the *100%* of *AccRepair* for all subjects (except *P20*) might be that the search algorithm is very accurate to find the maximum floating-point error in a small search space. A higher level of error thresholds needs an approximation function with more pieces to reduce the error. Then the search space is partitioned more fine-grained, which also decreases the difficulty to find a maximum floating-point error in the smaller search space and increases the accuracy of the search algorithm. We have also repeated our experiments many times with different random seeds, and the accuracy of repair keeps *100%*. However, we can not guarantee the accuracy of repair be always *100%* for all programs.

Influence on functional correctness According to Theorem 4.1 and Theorem 4.2, in principle, our approach can produce a repair semantically equivalent to the mathematical function by setting the error threshold ε equal to zero (in the worst case by storing the mathematical result for each floating-point input inside I_{err}). Besides, for a numerical program involving ill-conditioned problems, the original implementation already affected the functional correctness, and rewrite-based approaches are not fit for fixing it. However, as a dynamic analysis method, we cannot guarantee the soundness of our approach unless we take an exhaustive search (i.e., testing all inputs).

Form of approximation An interesting question is why we choose the piecewise quadratic function to approximate a mathematical function. Accurately implementing approximations is critical for approximating mathematical functions. Using quadratic approximations, we can have strategies (in Sect 4.3) to make the subtraction operations (in approximations) exact and thus can keep high accuracy during the implementation (while other approximations may not have such properties). Moreover, we choose a simple way to solve the problem and our experimental results show that our approach is effective on most subjects.

7 RELATED WORK

Floating-point error detection Since the work of Benz et al. [2012], the study of dynamically detecting floating-point error has gained momentum. Benz et al. [2012] developed a tool called FpDebug, which is built based on MPFR [Fousse et al. 2007] and Valgrind [Nethercote and Seward 2007], and can do a shadow execution of the original program in a higher precision to detect floating-point errors for every instruction. Based on FpDebug, Zou et al. [2015] proposed a detection approach called LSGA to search for input that can trigger possible maximum floating-point error in a numerical program. Recently, Herbgrind, which is a similar tool of FpDebug, is developed by

Sanchez-Stern et al. [2018]. Besides supporting similar functionalities of FpDebug, Herbgrind can find the root cause of floating-point error and extract the corresponding floating-point expressions. However, the above approaches or tools are all based on the assumption that the semantics of floating-point code in the higher precision is closer to the semantics of mathematical function. Hence they could not deal with the precision-specific operations [Wang et al. 2016] and may introduce unexpected errors. Aware of the precision-specific operations in numerical program, Yi et al. [2017b] verified that many of high floating-point errors in GSL reported by Zou et al. [2015] are in fact false alarms. Yi et al. [2017b] also proposed a new search algorithm called EAGT with a new fitness function from error analysis.

Compared with previous detecting methods, as far as we know, there is no existing work combining condition number and MCMC for finding the input triggering maximum error. Moreover, our detecting method not only uses the DEMC algorithm to search the input that can trigger the maximum floating-point error (like existing detecting methods [Chiang et al. 2014] [Zou et al. 2015] [Yi et al. 2017b]), but also uses the PTB algorithm to localize inputs that can trigger floating-point errors higher than a given repair criterion.

Automated improving accuracy of floating-point expressions Tools for automated improving accuracy of floating-point expressions can be classified by the underling analytic methods (static or dynamic). Static analysis of floating-point code tries to provide a sound bound of errors but may have low precision due to too conservative over-approximations and may have limited scalability. In contrast, dynamic analysis can scale for large program and find exact floating-point errors during dynamically executing of numerical program but cannot guarantee a sound bound of errors for a given input interval.

Static tools: Salsa [Damouche and Martel 2018] is a tool constructed under sound abstract interpretation [Cousot and Cousot 1977] and uses mathematical equivalent transformation [Damouche et al. 2017] to improve the accuracy of floating-point programs. Salsa can analyze numerical programs with loops and functions, but cannot deal with complex data structures like arrays, function pointers, library function calls, etc.

Compared with Salsa, our work is based on dynamic analysis. Thus, we cannot guarantee a sound bound of errors for a given input intervals like Salsa, but our experimental results show that we produce a high accuracy rate (100%) of repair to reduce high floating-point error to satisfy a given error threshold over a large amount of floating-point inputs.

Dynamic tools: Herbie [Panchekha et al. 2015] is a tool for automatically improving the accuracy of floating-point expressions. Like Salsa, Herbie uses mathematical rewriting to improving accuracy. To apply Herbie to numerical programs, the same authors of Herbie develop the tool Herbgrind [Sanchez-Stern et al. 2018] to find root cause of floating-point error and extract the corresponding floating-point expressions that can be analyzed by Herbie. The combination of Herbie and Herbgrind is called HBG in our paper and its detailed implementation is introduced in §5.2. AutoFP [Yi et al. 2017a] is a similar tool of HBG, but is constructed based on FpDebug [Benz et al. 2012] and Herbie. AutoFP tries to divide program into blocks to improve accuracy in every block to decrease the complexity of analysis. Although Yi et al. [2017a] first try to use AutoFP to repair high accuracies in numerical programs, AutoFP uses FpDebug with Herbie and thus is quite similar as HBG (Herbgrind with Herbie). Moreover, AutoFP does not support numerical programs with complex data structures in GSL.

Compared with the above work, our work targets at a different goal and uses different methods. Concerning the target, we focus on automatically repair high floating-point error under a given threshold rather than improving accuracy without a given criterion or target. Concerning methodology, our approach uses the linear approximation with error compensation to derive an approximation of mathematical function instead of using the mathematical rewriting to rearrange floating-point expressions in original numerical program.

Automated program repair A large and growing body of literature has investigated the automated program repair. Patch generation is the core part of the whole automated repair process. Based on the method of patch generation, most automated program repair (APR) approaches can be classified into two types below:

Search-Based APR: Weimer et al. [2009] proposed GenProg which first uses the genetic algorithm to search possible useful sentences to generate patches. After the opening work of Weimer et al. [2009], other search algorithms, e.g., random search in [Qi et al. 2014], have been used for repairing, and methods [Long and Rinard 2015][Long and Rinard 2016] for selecting candidate patches have also been well studied.

Semantic-Based APR: Nguyen et al. [2013] proposed SemFix which first uses the input and output of test case to generate program contract and then uses program synthesis to generate patches. Methods of semantic-based APR have been applied for repairing specific types of bugs, such as conditional bugs [DeMarco et al. 2014][Xuan et al. 2017][Durieux and Monperrus 2016], infinite loop [Marcote and Monperrus 2015], and assignments [Gopinath et al. 2011]. Methods of semantic-based APR [Mechtaev et al. 2015][Mechtaev et al. 2016] have also been used for repairing general bugs by using some simplified strategies.

Compared with other ARP methods, our approach focuses on repairing high floating-point errors which is also considered as accuracy bugs [Di Franco et al. 2017].

8 CONCLUSION AND FUTURE WORK

In this paper, we have proposed a novel approach to automatically repair high floating-point errors in numerical libraries. Our approach provides a detecting method including two algorithms (i.e., DEMC and PTB, see §4.1) to detect high floating-point errors and a repairing method based on deriving an approximation of the mathematical function to generate patch to satisfy a given repair criterion. We developed a prototype tool called AutoRNP for repairing high floating-point errors in numerical programs of GSL. Experimental results show that our tool successfully repair 19 of 20 numerical programs in GSL (with 100% accuracy).

While our approach is presented for numerical programs with one floating-point input, for future work, we plan to extend our approach to be generally applicable to numerical programs with multiple floating-point inputs.

ACKNOWLEDGMENTS

We thank Pavel Panchekha for his discussions on using Herbie and Herbgrind, and Zhengfeng Yang for his helpful suggestions on this work. We also thank the anonymous reviewers for their valuable comments. This work is supported by the National Key R&D Program of China (No. 2017YFB1001802), and the National Natural Science Foundation of China (Nos. 61672529, 61872445, 61502015).

REFERENCES

Milton Abramowitz. 1974. *Handbook of Mathematical Functions, With Formulas, Graphs, and Mathematical Tables,*. Dover Publications, Inc., New York, NY, USA.

Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. 2003. An Introduction to MCMC for Machine Learning. *Machine Learning* 50, 1 (01 Jan 2003), 5–43. https://doi.org/10.1023/A:1020281327116

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 &#38; Applications (OOPSLA '13).* ACM, New York, NY, USA, 817–832. https://doi.org/10.1145/2509136.2509526

Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A Dynamic Program Analysis to Find Floating-point Accuracy Problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 453–462. https://doi.org/10.1145/2254064.2254118

Florian Cajori. 1911. Horner's method of approximation anticipated by Ruffini. *Bull. Amer. Math. Soc.* 17, 8 (1911), 409–414. https://doi.org/10.1090/S0002-9904-1911-02072-9

Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient Search for Inputs Causing High Floating-point Errors. *SIGPLAN Not.* 49, 8 (Feb. 2014), 43–52. https://doi.org/10.1145/2692916.2555265

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. ACM, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

Nasrine Damouche and Matthieu Martel. 2018. Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs. In *Automated Formal Methods (Kalpa Publications in Computing)*, Natarajan Shankar and Bruno Dutertre (Eds.), Vol. 5. EasyChair, 63–76. https://doi.org/10.29007/j2fd

Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. 2017. Numerical Accuracy Improvement by Interprocedural Program Transformation. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES '17)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/3078659.3078662

Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic Repair of Buggy if Conditions and Missing Preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA '14)*. ACM, New York, NY, USA, 30–39. https://doi.org/10.1145/2593735.2593740

Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A Comprehensive Study of Real-world Numerical Bug Characteristics. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. IEEE Press, Piscataway, NJ, USA, 509–519. http://dl.acm.org/citation.cfm?id=3155562.3155627

Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *Proceedings of the 11th International Workshop on Automation of Software Test (AST '16)*. ACM, New York, NY, USA, 85–91. https://doi.org/10.1145/2896921.2896931

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, Article 13 (June 2007). https://doi.org/10.1145/1236463.1236468

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 '15)*. ACM, New York, NY, USA, 639–654. https://doi.org/10.1145/2814270.2814317

Zhoulai Fu and Zhendong Su. 2017. Achieving High Coverage for Floating-point Code via Unconstrained Programming. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, New York, NY, USA, 306–319. https://doi.org/10.1145/3062341.3062383

David Goldberg. 1991. What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Comput. Surv.* 23, 1 (March 1991), 5–48. https://doi.org/10.1145/103162.103163

Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-based Program Repair Using SAT. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software (TACAS'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 173–188. http://dl.acm.org/citation.cfm?id=1987389.1987408

Fredrik Johansson et al. 2013. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18)*. `http://mpmath.org/`.

William Kahan. 1996. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE* 754, 94720-1776 (1996), 11.

Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2017. On Automatically Proving the Correctness of Math.H Implementations. *Proc. ACM Program. Lang.* 2, POPL, Article 47 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158135

Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. ACM, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811

Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. *SIGPLAN Not.* 51, 1 (Jan. 2016), 298–312. https://doi.org/10.1145/2914770.2837617

Sebastian R. Lamelas Marcote and Martin Monperrus. 2015. Automatic Repair of Infinite Loops. *CoRR* abs/1504.05078 (2015). arXiv:1504.05078 http://arxiv.org/abs/1504.05078

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 448–458. http://dl.acm.org/citation.cfm?id=2818754.2818811

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. https://doi.org/10.1145/1273442.1250746

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 772–781. http://dl.acm.org/citation.cfm?id=2486788.2486890

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 '15)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2737924.2737959

Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. ACM, New York, NY, USA, 254–265. https://doi.org/10.1145/2568225.2568254

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 '18)*. ACM, New York, NY, USA, 256–269. https://doi.org/10.1145/3192366.3192411

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic Optimization of Floating-point Programs with Tunable Precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/2594291.2594302

Pat H Sterbenz. 1973. *Floating-point computation.* Prentice Hall, Englewood Cliffs, N.J.

Rainer Storn and Kenneth Price. 1997. Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization* 11, 4 (01 Dec 1997), 341–359. https://doi.org/10.1023/A:1008202821328

Enyi Tang, Xiangyu Zhang, Norbert Th. Muller, Zhenyu Chen, and Xuandong Li. 2017. Software Numerical Instability Detection and Diagnosis by Combining Stochastic and Infinite-Precision Testing. *IEEE Transactions on Software Engineering* 43, 10 (Oct 2017), 975–994. https://doi.org/10.1109/TSE.2016.2642956

David J. Wales and Jonathan P. K. Doye. 1997. Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms. *The Journal of Physical Chemistry A* 101, 28 (1997), 5111–5116. https://doi.org/10.1021/jp970984n

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 (FSE '16)*. ACM, New York, NY, USA, 619–630. https://doi.org/10.1145/2950290.2950355

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Softw. Eng.* 43, 1 (Jan. 2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2017a. Automated Repair of High Inaccuracies in Numerical Programs. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME '17)*. 514–518. https://doi.org/10.1109/ICSME.2017.45

Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2017b. Efficient Global Search for Inputs Triggering High Floating-Point Inaccuracies. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC '17)*. 11–20. https://doi.org/10.1109/APSEC.2017.7

Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. 2015. A Genetic Algorithm for Detecting Significant Floating-point Inaccuracies. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 529–539. http://dl.acm.org/citation.cfm?id=2818754.2818820