Efficient Global Search for Inputs Triggering High Floating-Point Inaccuracies

Xin Yi, Liqian Chen, Xiaoguang Mao, Tao Ji
College of Computer, National University of Defense Technology
Changsha 410073, China
Email: {yixin09, lqchen, xgmao,taoji}@nudt.edu.cn

Abstract—**Floating-point rounding errors are pervasive when using numerical code to implement the real arithmetic algorithm. In particular, high floating-point inaccuracies may cause serious problems once being triggered. Hence, a testing method that can find concrete test cases to trigger high floating-point inaccuracies, is quite helpful to aid debugging and reduce high inaccuracies. Recently, two testing approaches have been proposed to find inputs triggering high floating-point inaccuracies in numerical programs: Locality-Sensitive Genetic Algorithm (LSGA) and Binary Guided Random Testing (BGRT). However, experiments show that LSGA may result in a high rate of false alarm while BART may easily fall into a local maximum when the search space is large. In this paper, we propose a novel testing approach to trigger high floating-point inaccuracies in numerical code. The main idea is utilizing heuristic rules drawn from error analysis to guide the process of global search of test cases. Comparative experiments with the random and BGRT methods are conducted on benchmarks including real-world scientific programs. Experimental results show that our approach can efficiently find inputs that trigger higher floating-point inaccuracies in 11 of 12 real-world programs (especially for programs whose input space are large) and have better stability.**

I. Introduction

In numerical code, floating-point inaccuracies are inevitable due to the rounding errors since floating-point number uses finite precision to represent the real number. High floating-point inaccuracies may lead to serious software failures and even disastrous results. Known examples include the Vancouver Stock Exchange event [1] and the sinking of the Sleipner A offshore platform [2].

Several methods [3][4][5], have been proposed to estimate the worst-case bounds of errors in numerical code with respect to the given input domain. However, the bounds of errors provided by static analysis are often too conservative [6].

Besides, the accuracy estimation can not provide the direct information to help debugging. In contrast, the testing method which can find concrete test cases that trigger high inaccuracies is a better choice for aiding debugging. Recently, two testing approaches have been proposed to find inputs triggering high floating-point inaccuracies in numerical code: Locality-Sensitive Genetic Algorithm (LSGA) [7] and Binary Guided Random Testing (BGRT) [6].

LSGA uses the genetic algorithm with meta-heuristic principles to automatically generate test inputs that can trigger high inaccuracies. We verified the 18 bugs that were detected by LSGA in [7] and found that 16 of 18 bugs were false alarms in the sense of comparing with the expected mathematical output of tested program. The high rate (88.8%) of false alarm is due to the fact that LSGA may introduce extra errors when so-called precision-specific operations [8] are not handled correctly, which is also pointed out in their successive work [8] that focuses on fixing the precision-specific operations. BGRT divides the input domain into two parts of the same size each time, and iteratively chooses the part that has the higher floating-point error among the results of sampling test. Apparently, this method has a fast velocity to reduce the search space (half after each iteration). However, due to the incomplete sampling, the part that in fact contains exact higher errors may be discarded, which lead to a local maximum.

Following the above two testing methods [6][7], we also consider the problem of triggering high floating-point inaccuracies in numerical programs as a search problem. To improve the search efficiency, there exits three challenges:

- **Finding effective heuristic rules**. Heuristic rules are used to guide a testing method to find an input that maximizes the error of the output. Effective heuristic rules should consider the general situation. The heuristic rules of LSGA method that are drawn from empirical analysis but without considering precision-specific operations are not generic and may lead to a high rate of false alarm.
- **Dealing with large search space**. Under the same sampling rate, the greater the search space, the more likely to lead to local maximization. The BGRT method, which discards half search space each iteration, would easily lead to local maximum especially for large search space. Proper handling of huge search space requires selecting an appropriate search algorithm and increasing the sampling rate.
- **Reducing time overhead**. In order to measure the accuracy of outputs of a numerical program, we need to calculate the real (mathematical) outputs of the program, which leads to the main time overhead for testing methods. Both LSGA and BGRT cost a lot of time in calculating real outputs. Reducing time overhead needs to reduce the number of calculations of real outputs.

To cope with these challenges, we propose a new method called EAGT (Error Analysis Guided Testing), which tries to provide generic heuristic rules drawn from error analysis and global search with high sampling rate while reducing time overhead of testing by leveraging efficient approximate calculation. Unlike the LSGA algorithm which draws heuristic rules from empirical analysis, we draw the heuristic rules from the forward and backward errors analysis. Moreover, we do not search iteratively like BGRT, instead, we search all the input space to decrease the possibility of local maximum and use approximate calculations to quickly get the intermediate results that are then used to guide the search.

We have conducted comparative experiments of EAGT with BGRT and random (RAND) testing methods on benchmarks including real-world scientific programs. Experimental results show that 1) EAGT, in most cases, can detect higher floating-point inaccuracies than BGRT and RAND; 2) Compared with BGRT and RAND, EAGT is more stable and more efficient especially for program whose input spaces is large.

In summary, this paper makes the following contributions:

- We propose a new searching approach namely EAGT, based on the heuristic rules obtained from error analysis. The main idea is using effective heuristic rules to guide the search process and leveraging approximate calculation to speed up global search, which can significantly improve the efficiency of testing method especially when the search space is large.
- We have conducted experiments on basic polynomial functions and numerical programs from GNU Scientific Library (GSL). Experimental results show that our approach can efficiently find inputs that trigger higher floating-point inaccuracies in 11 of 12 real-world programs (especially for programs whose input space are large) and have better stability than BGRT and RAND.

The rest of this paper is organized as follows. Section II gives the background on floating point representation and error analysis. Section III shows empirical study of LSGA and BGRT. Section IV introduces our approach. Section V shows our experiment setting and research questions. Section VI provides experimental results and answers research questions. Section VII gives discussion and future work. Section VIII concludes.

II. BACKGROUND

The section introduces basic background of floating-point format and error.

A. Floating-Point Format

According to the IEEE-754 Standard [9], a floating-point number can be represented by

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

where $S \in \{0, 1\}$ represents the sign, $M = m_0.m_1m_2...m_\mathbf{p}$ is the significand (also called the mantissa), where $.m_1m_2...m_\mathbf{p}$ represents a **p**-bit fraction and m_0 is the hidden bit without need of storage, and $E = e - bias$ means exponent, is a e-bit signed integer and $bias = 2^{e-1} - 1$. Table I shows the bits of the sign, exponent, and mantissa of 32-bit single and 64-bit double precision floating-point according to the IEEE-754 Standard.

TABLE I
SINGLE AND DOUBLE PRECISION FLOATING-POINT

	sign	mantissa	exponent
Single	1	8	23
Double	1	11	52

B. Error of Floating-Point

Based on the format of floating-point, floating-point numbers can not exactly represent all real numbers. The conversion of the floating-point to the real number unavoidably introduces rounding errors for some values. Fig. 1 shows the conversion intuitively.

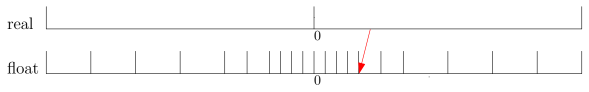


Fig. 1. Conversion from real to float

As shown in Fig. 1, the rounding error less than one ULP (the unit in the last place [10]) under the conversion. The ULP can be simply regarded as the distance between a floating-point number and its the next floating-point number. Due to the fixed bits in the mantissa, with the increasing of exponent value, the ULP value also bigger.

The absolute error and relative error are two common indicators to evaluate the error of floating-point program. For a real function $f(x)$, we use $f_p(x)$ as its corresponding floating-point program. The formulas (2) and (3) show the absolute error and relative error.

$$Absolute\_error = |f(x) - f_p(x)| \tag{2}$$

$$Relative\_error = \frac{|f(x) - f_p(x)|}{|f(x)|} \tag{3}$$

Absolute error and relative error are defined from the mathematical point of view, so they both are ill-suited to floating-point error measure [11]. In this paper, we use a standard method to measure the floating-point error, which is also used in [11] [12] [13]. The method can be illuminated by the formulas (4) and (5).

$$ulp_{error}\{f(x), f_p(x)\} = \frac{|f(x) - f_p(x)|}{|ulp(fl(f(x))|} \tag{4}$$

$$bit_{error}\{f(x), f_p(x)\} = log_2\{ulp_{error}\{f(x), f_p(x)\}\} \tag{5}$$

In formula (4), $fl(f(x))$ returns the floating-point value of $f(x)$, and $ulp(fl(f(x))$ returns the ULP value of $fl(f(x))$. In this way, we can evaluate the error by the significant bits through the $log_2(x)$ function, as shown in formula (5).

III. EMPIRICAL STUDY OF LSGA AND BGRT

A. LSGA Approach

Zou et al. [7] proposed the LSGA approach which had conducted experiments over GSL programs and achieved remarkable results. According to their experimental results, LSGA reports 18 potential bugs in GSL programs.

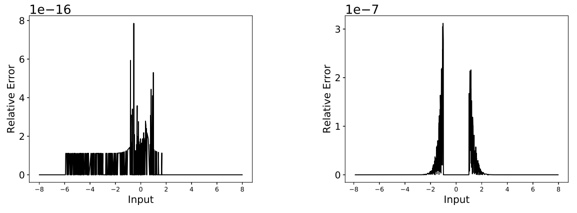
TABLE II
TEST RESULTS: LSGA VS MPMATH

Program	Relative Error		
	LGSA	mpmath	L/m
gsl_sf_airy_Ai_deriv	1.54E+06	2.29E-07	6.72E+12
gsl_sf_airy_Ai_deriv_scaled	1.54E+06	2.29E-07	6.72E+12
clausen	5.54E-02	3.74E-02	1.48E+00
eta	9.58E+13	2.38E-15	4.02E+28
exprel_2	2.85E+00	5.25E-12	5.43E+11
gamma	1.07E-02	2.17E-16	4.94E+13
synchrotron_1	5.35E-03	5.95E-14	8.99E+10
synchrotron_2	3.67E-03	1.30E-13	2.83E+10
zeta	9.58E+13	2.04E-15	4.70E+28
zetam1	1.42E-02	2.50E-15	5.67E+12
bessel_Knu	6.08E-03	2.10E-15	2.90E+12
bessel_Knu_scaled	6.08E-03	2.10E-15	2.90E+12
beta	9.21E-03	9.75E-16	9.44E+12
ellint_E	8.92E-03	1.56E-16	5.70E+13
ellint_F	8.79E-03	1.33E-16	6.62E+13
gamma_inc_Q	1.36E+13	3.62E-05	3.75E+17
hyperg_0F1	5.80E+06	7.33E+49	7.91E-44
hyperg_2F0	4.35E-03	1.62E-13	2.69E+10

To check those potential bugs, we need to get the output produced by the real arithmetic function corresponding to the original program. FPDebug [14] can be used to supply approximate real arithmetic output for LSGA. However, FPDebug directly increases precision on all floating-point operations of tested programs, which may introduce extra errors for some precision-specific floating-point operations [8] and break the semantics of the original program. For example, the precision-specific operation $(x + n) - n$ $(where\ n = 6755399441055744.0)$ is used to round x to integer under the 64-bit floating-point arithmetic. n is a special number that only works for 64-bit floating-point arithmetic, and increasing the precision (e.g., to 128-bit) for this operation will save the decimal part of x, which breaks the semantic of the operation and introduces extra error (i.e., the value of decimal part of x).

To avoid the possible extra errors, we use programs from mpmath [15] which is a free Python library for real and complex floating-point arithmetic with arbitrary precision, as the real (mathematical) functions to calculate real outputs. For example, we use the $zeta$ program in mpmath as the real arithmetic version of gsl_sf_zeta program in GSL. By directly using programs in mpmath, we can keep the real arithmetic semantics of the programs that are tested while without introducing extra errors for precision-specific operations.

As our experimental resuls shown in TABLE II, except for $clausen$ and $hyperg\_0F1$, the relative errors returned by LSGA are ten orders of magnitudes greater than mpmath at least (see the "L/m" column which shows the ratio of relative errors given by LSGA and mpmath), which means that only 2 of 18 inputs that produced by LSGA indeed trigger high inaccuracies and the false alarm rate of LSGA is around 88.8%.



(a) Error reported by mpmath (b) Error reported by FPDebug

Fig. 2. Error distribution of program erf

The heuristic rules of LSGA are extracted from the empirical analysis. There are five heuristic rules that are used to guide the LSGA in [7]:

(a). Both exponents and the significands have great impact on the relative error;
(b). Exponents that invoke large relative errors stay in a small interval;
(c). Only small parts of exponents invoke the large error;
(d). Near the large error would have error higher than average;
(e). The exponents that lead to significant errors are likely near 1023.

We investigate the five heuristic rules from two aspects: One is whether the results that are used to do the empirical analysis are not right from a certain point of view; Another is whether there exist counterexamples that disobey some of those heuristic rules.

For the first aspect, we retest the erf program which is used as one of the examples to get those heuristic rules in [7]. We get the relative error by using mpmath and FPDebug respectively, and the results are shown in Fig. 2. The maximum error reported by FPDebug reaches the order 1e-7, but the error reported by mpmath is just under the order 1e-15. The difference is due to the fact that FPDebug can not handle precision-specific operations correctly, which introduces extra errors. The difference shows that those heuristic rules are somehow built on unreliable empirical analysis.

For the second aspect, we try to find counterexamples that do not fit in some of those heuristic rules. In the community of GSL, we find bug reports[1] that show many functions do not obey some of those heuristic rules. Those bug reports exhibit that high floating-point inaccuracies would occur when the input value increases beyond the value 823549.6645 in those functions, which conflict with three heuristic rules (b, c, e) that implicate the errors stay in a small part and the exponents leading to large errors are near the value 1023. And our experimental results shown in Fig.7(a) (Section VI) also indicate cases not fitting for the three heuristic rules.

From the above analysis, we see that not all heuristic rules of LSGA are generic and fit for general cases. In our experimental results, the high false alarm rate (88.8%) of LSGA also supports this argument.

[1]https://savannah.gnu.org/bugs/?45746

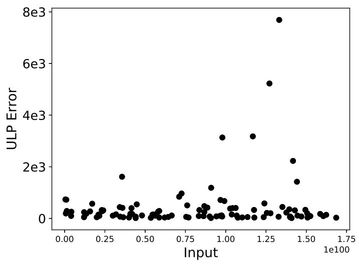


Fig. 3. Error distribution of program gsl_sf_bessel_J1

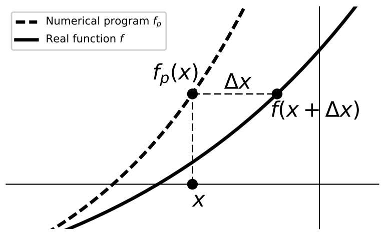


Fig. 4. Backward error

B. BGRT Approach

The BGRT method proposed by Chiang et al. [6] is a straightforward binary search method. BGRT divides the input domain into two parts of same size each time, and iteratively chooses the part that has the higher floating-point error in the results of sampling test inputs. Apparently, when the input space is huge, BGRT is easy to fall into a local maximum due to the incomplete sampling of input domain. The only way to get out from the local maximum is to reset the search domain to the initial domain [6], which would waste more time and still can not guarantee not getting into another local maximum after reset. For illustrating the local maximum of BGRT algorithm intuitively, we test the BGRT algorithm on the function $gsl\_sf\_bessel\_J1$ which is a GSL program for implementing the 1 order regular cylindrical Bessel function. Fig. 3 shows the distribution of maximum errors that BGRT finds in 100 trials. The distribution of maximum errors are dispersing, and only few trials result in several inputs that reach the relatively higher inaccuracies. The results illustrate that the BGRT algorithm easily terminates with a local maximum when the input domain is large.

Theoretically, the local maximum is hard to avoid because the search is not exhaustive, and the BGRT which throws out half input domain each iteration would easily lead to a local maximum.

IV. APPROACH

In this section, we illustrate the error analysis and the heuristic rules that are used in our approach. After that, we introduce our Error Analysis Guided Testing (EAGT) approach.

A. ERROR ANALYSIS

Error analysis can be divided into two kinds: forward and backward error analysis. Forward error analysis has been extensively studied and widely applied to floating-point, such as in dynamic testing [14], precision tuning [12] and static analysis [16] [17].

In contrast, backward error analysis is seldom used. In this paper, both backward and forward error analysis results are used for drawing the heuristic rules.

- **Forward Error.** For calculating the forward error, we just concern the error of the output of the origin program rather than the intermediate error for each instruction in the program. Therefore, the forward error is only calculated one time for each input, and the formula (3) represents the standard forward error of the origin program. In this paper, we used formulas (4, 5) instead of (3), which fit better for the floating-point arithmetic.
- **Backward Error.** In Section II-B, we use $f_p(x)$ to approximate $f(x)$. For backward error, we would like to find the smallest Δx such that
$$f_p(x) \simeq f(x+\Delta x)$$
Here, we mean that $f_p(x)$ equals approximately to the exact value of $f(x+\Delta x)$. Δx (or $\frac{\Delta x}{x}$) is called the backward error. Fig. 4 intuitively shows the concept of the backward error.

Our next analysis is based on the following assumption: the function f is smooth around the neighbourhood of x. This assumption is reasonable when $\delta = \frac{\Delta x}{x}$ is small. According to the analysis of Fu et.al [18] for numerical code, we find that the backward error is usually small. Under the assumption above, we reconstruct the forward error by combining with backward error and Taylor expansion.

$$f(x+\delta\cdot x) \approx f(x) + f'(x)(\delta\cdot x) + \Theta(\delta^2) \tag{6}$$

$$\Rightarrow |f(x) - f(x+\delta\cdot x)| \approx |f'(x)(\delta\cdot x)| \tag{7}$$

$$\Rightarrow \left|\frac{f(x)-f(x+\delta\cdot x)}{ulp(fl(f(x)))}\right| \approx \left|\frac{\delta\cdot x\cdot f'(x)}{ulp(fl(f(x)))}\right| \tag{8}$$

$$\Rightarrow \left|\frac{f(x)-f(x+\delta\cdot x)}{ulp(fl(f(x)))}\right| \approx \left|\frac{\Delta x\cdot f'(x)}{ulp(fl(f(x)))}\right| \tag{9}$$

According to Formulas (4) and (9), we have

$$ulp_{error}\{f(x), f_p(x)\} \approx \left|\frac{\Delta x\cdot f'(x)}{ulp(fl(f(x)))}\right| \tag{10}$$

By replacing Δx with $(n+\lambda)\cdot ulp(x)$ where $n \in N$ is a natural number and $\lambda \in [0,1]$, we get

$$ulp_{error}\{f(x), f_p(x)\} \approx \left|\frac{(n+\lambda)\cdot ulp(x)\cdot f'(x)}{ulp(fl(f(x)))}\right| \tag{11}$$

The substitution above is valid when δ is small, which means the distance between $x' = (x+\Delta x)$ and Δx can be represented by $n\cdot ulp(x) + \lambda\cdot ulp(x)$ when x' is nearby of x, as depicted in Fig. 5.

After reconstructing the forward error, we can divide the forward error into two parts:

- The index of backward error: $B_i = |(n+\lambda)|$
- The condition number that is represented by floating-point ULP: $C_{fp} = |f'(x)| \cdot \left|\frac{ulp(x)}{ulp(fl(f(x)))}\right|$

Then, $ulp_{error}(f(x), f_p(x))$ can be approximated as

$$ulp_{error}(f(x), f_p(x)) \approx B_i \times C_{fp} \tag{12}$$

According to the definition of B_i, the value of B_i that depends on the implement action of f_p can not be calculated directly. We think trying to find the large B_i is unrealistic, because the implemented program may be too complicated and hard to predict the law of the distribution of B_i. This point is also supported by the experiments of Fu et.al [18]. Their experiments use the Monte Carlo Markov Chain (MCMC) techniques to estimate the backward error, but waste much time even in a small input range for some basic functions, such as sin, cos, sqrt, etc.

Since the value of B_i is unfortunately hard to calculated, we decide to use the distribution of C_{fp} to estimate the possible high floating-point inaccuracies and propose the following heuristic rules that are used in our approach:

- High floating-point inaccuracy has a greater probability of having a large value of C_{fp}. According to the formula (12), the high inaccuracies may be more likely caused by the large value of C_{fp} than the small one.
- The maximum value of C_{fp} does not guarantee the maximum floating-point error. According to the formula (12), B_i also affects the floating-point error of output. Moreover, according to the definition of B_i and C_{fp}, the value of C_{fp} is independent of f_p while B_i depends on f_p. Thus, the maximum value of C_{fp} dose not mean the maximum value of B_i.

B. EAGT Approach

According to our heuristic rules, we need to quickly calculate the value of C_{fp} to find the large C_{fp} to guide the search. In fact, we do not need to know the exact value of C_{fp}, since we just use the value of C_{fp} as a guide and do not need the value to calculate the final result, which means that an approximate value is acceptable. The approximate value of $f'(x)$ can be calculated by a derivative evaluated function on $f_p(x)$. The derivative evaluated function is supplied by many scientific function libraries. The value of $|ulp(fl(f(x)))|$ can also be directly obtained by approximating $|ulp(f_p(x))|$, because in most situations the ULP of outputs should be approximate identical for $f(x)$ and $f_p(x)$, under the assumption

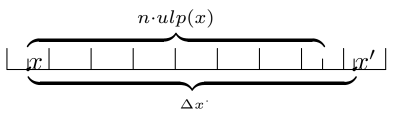


Fig. 5. Distance between x' and x

Algorithm 1 Error Analysis Guided Testing

```
Input: $f_p, C_{init}, K$
Output: $error\_list$
 1: $tmp\_list \leftarrow [\ ]$
 2: $error\_list \leftarrow [\ ]$
 3: while has resources do
 4:     $Confs \leftarrow Partition(C_{init})$
 5:     for $c \in Confs$ do
 6:         $(l_{Appro(C_{fp})}, l_x) \leftarrow C_{fp}\_Eval(f_p, c)$
 7:         $(err_i, x_i) \leftarrow Error\_Eval(f_p, l_x)$
 8:         $tmp\_list.append((err_i, x_i))$
 9:     end for
10:     $tmp\_list \leftarrow SortByError(tmp\_list)[0 : K]$
11:     for $(err, x_i) \in tmp\_list$ do
12:         $Conf_{x_i} \leftarrow Reconf(x_i)$
13:         $(max\_err, x_t) = Error\_Eval(f_p, Conf_i)$
14:         $error\_list.append((max\_err, x_t))$
15:     end for
16: end while
17: $error\_list \leftarrow SortByError(error\_list)$
18: return $error\_list$
```

that the high errors stay in a small interval. In this way, the approximate value of C_{fp} can be represented by:

$$Appro(C_{fp}) = \left|Derivative(f_p(x), x) \cdot \frac{ulp(x)}{ulp(f_p(x))}\right| \tag{13}$$

After the conversion from C_{fp} to $Appro(C_{fp})$, the value of $f(x)$ is not needed, which means that we are free from the high precision calculation at the step of calculating C_{fp} value and save a lot of time.

Alg.1 shows our EAGT algorithm. The inputs of our algorithm include f_p, C_{init} which represents the input domain of the real function, and K that is used to trade off between computer resources and search stability. The algorithm does not directly output the max error that it find, but return an error list. Apparently, an error list can supply more information than just a single error. To produce the error list efficiently, our algorithm includes three main steps: partitioning, global search and fine search. We explain these three steps in detail as follows:

- **Partitioning:** The single line 4 in Alg. 1 completes this step. The function $Partition(C_{init})$ tries to divide the input domain C_{init} into a set of intervals $Confs$. We try to divide the input domain into a set of intervals such that each interval contain the same numbers of floating-point inputs to make sure the sampling rate is equal for each c in $Confs$. However, the fact that the interval $[-1, 1]$ contains half number of all floating-points numbers, which will desire massive tests and consume much time when the considered interval contains [-1,1]. In the function $Partition(C_{init})$, we consider $[-1, 1]$ as an independent interval to test and divide other part of input domain according to the numbers of contained floating-point numbers. In this way, we keep the sample rate the same for every $c \in Confs$ except $[-1, 1]$ and also save a lot of computer resources.

- **Global Search:** This step corresponds to lines 5-9 in Alg. 1. The function $C_{fp}\_Eval(f_p, c)$ samples inputs within the small input domain $c \in Confs$ and returns the list of $Appro(C_{fp})$ with maximum first together with its corresponding input list l_x. The function $Error\_Eval(f_p, l_x)$ calculates the error for the one hundred inputs in l_x and returns the maximum error (err_i) together with the corresponding input (x_i). After that, the tmp_list saves the (err_i, x_i) for the next fine search.
- **Fine Search:** The fine search correspond to line 11-16 in Alg. 1. Before the fine search, the $SortByError(tmp\_list)[0{:}k]$ function on line 10 would sort the tmp_list according to the error value of the element tuple (err_i, x_i) and return the first K elements, where the K value is set for the trade-off between stability and the consumption of computer resources. The closer the K value to the length of $Confs$, the more global information is saved, and also more computer resources are consumed, and vice verse. We set the K value according to the length of $Confs$ in experiments. The $Reconf(x_i)$ function in line 12 would generate a small interval $Conf_{x_i}$ around x_i. Next, the small interval is transferred to function $Error\_Eval(f_p, Conf_i)$ which would return the max error (max_err) for $Conf_{x_i}$ and also return the input x_t that triggers the max error. At last, (max_err, x_t) is saved in $error\_list$.

In our approach, we cope with the three challenges mentioned in Section I by studying backward and forward error to achieve general heuristic rules, using the approximate calculation of C_{fp} to reduce the time overhead, and global search to deal with the large search space problem.

V. EXPERIMENTAL DESIGN

To check the performance of our EAGT method, we conduct comparative experiments to compare EAGT with the existing BGRT method and the random method. We select BGRT to compare due to the fact that it is the only state-of-the-art high inaccuracy detecting approach with (open source) tool available. We can not compare with the LSGA method by experiments because its corresponding tool is not publicly available, and we do not know the detailed parameter setting of the LSGA approach. We have conducted empirical study of the LSGA approach in Section III and found that the LSGA approach may not fit for general cases, especially for programs involving precision-specific operations.

A. Research Questions

Our experimental evaluation seeks to address the following research questions:

RQ1: Whether EAGT can find higher floating-point inaccuracies with limited resources, compared to BGRT?

The situation is complicated for testing floating-point programs, and many factors need to consider, such as testing time, sampling rate and the size of input domain. EAGT is designed with the ability of global and fast search. BGRT is also fast but may easily get into a local maximum. It is hard to judge which method can find higher floating-point inaccuracies. RQ1 asks whether EAGT can find higher floating-point inaccuracies compared to BGRT under the same limitation.

RQ2: Which algorithm has better stability, BGRT or EAGT?

An approach with low stability needs more repetition of experiments to get a useful result, which will waste more resources. The stability should not be measured by just one factor. We measure the stability from two aspects of experimental results: the validity and the degree of dispersion. The validity is to measure whether the results are worth to do the comparison of stability. For example, if the floating point inaccuracies that are found by a method are close to zero, we consider the stability is not necessary to measure in this situation. The degree of dispersion is to measure the distribution of 100 trials data for each approach. We use the coefficient of variation as the index for the degree of dispersion. In this way, we measure the stability of EAGT and BGRT from the two above aspects, to answer RQ2.

TABLE III
SUBJECT PROGRAMS

	Program	Detail
Poly	Bell	$x^5 - (5/2) \cdot x^4 + (5/3) \cdot x^3 - (1/6) \cdot x$
	Bernoulli	$x^5 + 10x^4 + 25x^3 + 15x^2 + x$
	Chebyshev	$16x^5 - 10x^3 + 5x$
GSL funcitons	gsl_sf_airy_ai	Airy function Ai(z)
	gsl_sf_Chi	Hyperbolic cosine integral
	gsl_sf_Ci	Cosine integral
	gsl_sf_bessel_J0	0 order regular cylindrical Bessel function
	gsl_sf_bessel_J1	1 order regular cylindrical Bessel function
	gsl_sf_bessel_Y0	0 order irregular cylindrical Bessel function
	gsl_sf_bessel_Y1	1 order irregular cylindrical Bessel function
	gsl_sf_eta	Alternating zeta function
	gsl_sf_gamma	Gamma function
	gsl_sf_legendre_P2	Legendre polynomial of the 2nd order
	gsl_sf_legendre_P3	Legendre polynomial of the 3rd order
	gsl_sf_lngamma	Logarithm of the Gamma function

B. Subject Programs

Our subject programs include two parts: 3 basic polynomial functions and 12 real-world GSL functions. Table III shows our subject programs in details. We first investigate polynomial functions because the polynomial approximate calculation has been widely used in many real-world functions. For example, we find that 210 special functions of GSL include about more than 230 times calls of polynomial evaluation, which indicates more than one polynomial function call for each GSL function on average. Thus, conducting experiment on the polynomial functions is essential and meaningful.

The GSL functions have received much attention in acadamic [16][19][20], but most existing work mainly concerns the stability or exceptions of GSL functions, rather than

TABLE IV
EXPERIMENTAL RESULTS OF POLYNOMIALS

Program	Input Domain	Algorithm	Mean of Max. Error	Median of Max. Error	Avg. Time(s)	Ratio on RAND	Ratio on Time
Bell	[-1.7e+10,1.7e+10]	EAGT	8.79E+18	8.79E+18	9.16	4395231165240180000.00	959743071030621000.00
		BGRT	3.00E+00	3.00E+00	10.83	1.50	0.28
		RAND	2.00E+00	2.00E+00	8.75	1.00	0.23
Bernoulli		EAGT	7.85E+18	8.73E+18	9.25	3926022478645970000.00	848657406067286000.00
		BGRT	3.00E+00	3.00E+00	12.07	1.50	0.25
		RAND	2.00E+00	2.00E+00	15.63	1.00	0.13
Chebyshev		EAGT	3.49E+18	4.36E+18	8.78	1745360127545860000.00	397693201029284000.00
		BGRT	2.90E+00	3.00E+00	11.60	1.45	0.25
		RAND	2.00E+00	2.00E+00	11.27	1.00	0.18
Bell	[-1.7e+2,1.7e+2]	EAGT	7.90E+18	8.79E+18	3.60	550595760539442.00	2198123333497370000.00
		BGRT	4.40E+18	4.39E+18	26.44	306577332734652.00	166441622966288000.00
		RAND	1.44E+04	3.59E+03	30.00	1.00	478.39
Bernoulli		EAGT	8.73E+18	8.73E+18	4.00	168963659934863.00	2183720295054170000.00
		BGRT	8.73E+18	8.73E+18	30.14	168887691531787.00	289448904481964000.00
		RAND	5.17E+04	1.09E+04	28.16	1.00	1834.79
Chebyshev		EAGT	4.36E+18	4.36E+18	3.34	861440548603637.00	1307560388529970000.00
		BGRT	2.18E+18	2.18E+18	27.66	430902860576154.00	78906893937393700.00
		RAND	5.06E+03	2.28E+03	24.83	1.00	203.95

the inaccuracy in GSL functions. The LSGA algorithm [7] indeed takes the special functions of GSL as benchmarks to detect inaccuracies, but its results may have high false alarm rate according to our empirical analysis discussed in Section III. For the sake of generality, we choose the GSL functions randomly from each kind of functions.

C. Experimental Setup

To make the comparison more clear, we use the random search algorithm as a baseline to measure BGRT and EAGT. We implement the BGRT method basically on the top of its tool[2], but we strengthen the method by increasing the iteration times. We do not limit the number of iterations. In other words, the method is terminated only when the resources are exhausted or the output is not changed by increasing the number of iterations. The EAGT method is implemented according to Alg.1. To be more clear, we change the K value according to the size of input domain. If the size is small, all items would be saved for fine search. Otherwise, we set a fixed value to K. It is worth to note that K is a configurable parameter for users.

All experiments are run on an Ubuntu 14.04 machine with 3.4 GHz Intel Core i7-4770 CPU and 8 GB of memory. To avoid the possible extra errors, we use the corresponding programs from mpmath [15] as implementation of the real functions of the corresponding benchmarks. Because the execution time is not the same for all programs, we assign the corresponding execution time to each program. The number of sample points is determined by the size of input domain, and thus is adjusted for each iteration. Especially, we separately perform 100 trials to test every GSL program for each method. We only do 10 trials for each polynomial function, but we change the size of the input domain to study the effect of the search space size on test methods.

VI. EXPERIMENTAL RESULTS

Table IV and V show the experimental results[3]. The first and second columns list the program names and input domains of programs. We calculate error of program output according to formula (4). Mean and median of maximum errors are presented in the 4th and 5th column for each algorithm. The 6th column shows the average time of testing. The "Ratio on RAND" column (i.e., the 7th column) is calculated by the formula below:

$$Ratio\_on\_RAND = \frac{Mean\ of\ Max\ Error}{RAND's\ Mean\ of\ Max\ Error}$$

In the above formula, the mean of maximum error of RAND is used as the baseline to measure the results of other two methods. We obtain the value of ratio on time (i.e., the 8th column) by using the ratio of the mean of maximum error (i.e., the 4th column) to the average time (i.e., the 6th column).

Next, we use our experimental results to answer the two research questions posed in Section V-A.

[2]https://github.com/wfchiang/s3fp

[3]All our experimental data are online: http://github.com/yixin-09/APSEC2017-EAGT-results

TABLE V
EXPERIMENTAL RESULTS OF GSL FUNCTIONS

Program	Input Domain	Algorithm	Mean of Max. Error	Median of Max. Error	Avg. Time(s)	Ratio on RAND	Ratio on Time
gsl_sf_airy_ai	[-823549,102]	EAGT	8.95E+18	8.95E+18	24.04	90348.33	372263878892634000.00
		BGRT	8.68E+18	8.95E+18	76.76	87600.57	113050884847208000.00
		RAND	9.91E+13	2.73E+13	76.28	1.00	1298555927826.22
gsl_sf_Chi	[0,700]	EAGT	1.29E+15	1.56E+15	2.41	45291790879435.90	535278776903093.00
		BGRT	1.08E+15	1.56E+15	5.71	37693909139754.50	188512902815526.00
		RAND	2.85E+01	9.50E+00	5.32	1.00	5.36
gsl_sf_Ci	[0,823549]	EAGT	3.15E+18	2.58E+14	3.55	165551244091756000.00	885034073176814000.00
		BGRT	3.49E+17	3.00E+00	1.87	18393998265178700.00	186411196173681000.00
		RAND	1.90E+01	3.00E+00	4.60	1.00	4.13
gsl_sf_bessel_J0	[0,1.7e+100]	EAGT	1.35E+15	1.78E+15	34.63	67924474978.14	38908928842321.60
		BGRT	7.70E+02	1.37E+02	64.61	0.04	11.91
		RAND	1.98E+04	9.88E+03	66.25	1.00	299.41
gsl_sf_bessel_J1	[0,1.7e+100]	EAGT	2.93E+18	4.37E+18	36.71	108942352709279.00	79729190418993000.00
		BGRT	4.62E+02	1.73E+02	69.42	0.02	6.65
		RAND	2.69E+04	9.54E+03	70.79	1.00	379.56
gsl_sf_bessel_Y0	[0,1.7e+10]	EAGT	1.28E+15	2.22E+14	107.74	31035187712.44	11883117715957.80
		BGRT	2.66E+06	2.75E+05	107.95	64.43	24622.17
		RAND	4.13E+04	7.88E+03	112.61	1.00	366.33
gsl_sf_bessel_Y1	[0,1.7e+10]	EAGT	2.20E+14	5.38E+13	106.22	6033080292.74	2072312643189.83
		BGRT	2.83E+07	3.03E+05	127.28	776.93	222710.04
		RAND	3.65E+04	5.37E+03	116.47	1.00	313.26
gsl_sf_eta	[-168,100]	EAGT	1.42E+15	7.50E+14	22.37	15384764304.41	63275557061608.80
		BGRT	2.38E+14	5.69E+13	28.46	2589851536.93	8373249684598.18
		RAND	9.20E+04	2.54E+04	27.95	1.00	3291.66
gsl_sf_gamma	[-168,168]	EAGT	6.20E+02	6.24E+02	2.70	1.01	222.16
		BGRT	5.56E+02	6.26E+02	3.10	0.91	179.24
		RAND	6.12E+02	6.14E+02	2.56	1.00	238.85
gsl_sf_legendre_P2	[-1.7e+10,1.7e+10]	EAGT	8.94E+14	9.47E+14	4.87	894415114308460.00	183581655766050.00
		BGRT	1.02E+00	1.00E+00	0.98	1.02	1.04
		RAND	1.00E+00	1.00E+00	5.56	1.00	0.18
gsl_sf_legendre_P3	[-1.7e+10,1.7e+10]	EAGT	4.24E+18	4.37E+18	5.07	2117566448980710000.00	835611238470093000.00
		BGRT	1.95E+00	2.00E+00	1.19	0.98	1.63
		RAND	2.00E+00	2.00E+00	6.98	1.00	0.29
gsl_sf_lngamma	[0,1000]	EAGT	2.90E+03	2.90E+03	1.82	44.23	1595.81
		BGRT	3.03E+03	3.07E+03	2.09	46.21	1449.17
		RAND	6.57E+01	4.55E+01	2.27	1.00	28.89

A. *RQ1*

RQ1 asks which algorithm can find the higher inaccuracy with limited resources. To answer the question, we first discuss the experimental results of polynomial functions.

In Table IV, we can discover that the input domain is a main influence factor for the results: The values of "Mean of Max. Error" of EAGT and BGRT are almost the same when the input domains are of small size; The values of "Mean of Max. Error" of EAGT are huge while that of BGRT is close to zero when the input domains are of large size. However, the values of "Ratio on Time" which indicate the number of the errors found in each second is quite different for different methods, when the input domain is small. Considering the limited resources, EAGT has the larger value of "Ratio on Time" than BGRT and thus can find higher inaccuracies faster.

Now, we discuss the experimental results on GSL functions. In Fig. 6, we convert the value of "Mean of Max. Error" to bit error according to the Formula (5) for intuitive comparison. As shown in Fig. 6, EAGT can find higher errors than BGRT in the 11 of 12 GSL functions, and the results for the other one (i.e., the $gsl\_sf\_lngamma$ function) are almost the same. We can also see the strong connection between the effectiveness of BGRT and the size of input domain. For the functions with small input domain, such as gsl_sf_Chi, gsl_sf_eta and gsl_sf_gamma, BGRT can also find the high inaccuracies. However, for the functions with large input domain, BGRT

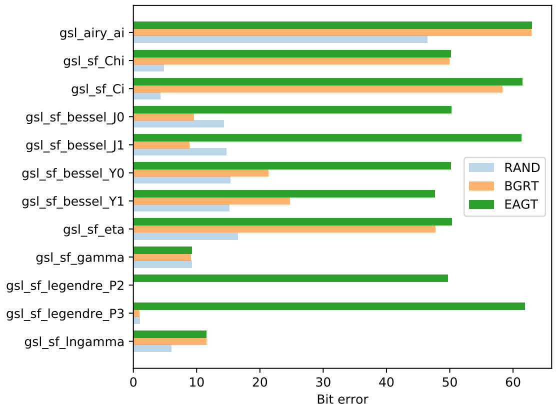
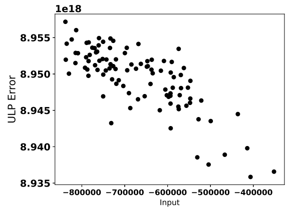
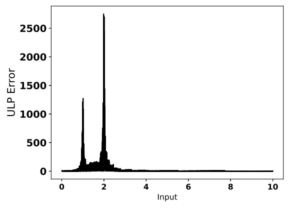


Fig. 6. Results of bit error



(a) gsl_sf_airy_ai



(b) gsl_sf_lngamma

Fig. 7. Error distribution

does not work well, even worse than the RAND, such as for $gsl\_sf\_bessel\_J1$ and $gsl\_sf\_legendre\_P3$, as shown in Table V. It is worth to notice that for all functions, EAGT has a larger value of "Ratio on Time" than the other two methods.

We also notice two exceptions in Table V. One is the $gsl\_sf\_airy\_ai$ function, for which the order of error values given by EAGT and BGRT are almost the same. The reason is that: for the function $gsl\_sf\_airy\_ai$, the high inaccuracies occur averagely with inputs that are around the lower bound of input domain as Fig. 7(a) shows. In this case, the high inaccuracies can easily detect by BGRT given enough iterations, but BGRT waste more time than EAGT (more than three times: $76.76s$ vs. $24.04s$). Another exception is the function $gsl\_sf\_lngamma$. As shown in Fig. 7(b), the inputs that can trigger the maximum error are few, which means more sample points are needed to hit the few inputs. The second exception can be dismissed by increasing the sampling rate of EAGT, but which would destroy the consistency of parameter setting of our experiment.

Answer for RQ1: In our experiment, for 11 of 12 GSL functions, EAGT can find higher floating-point inaccuracies than BGRT and RAND. And for the value of "Ratio on Time", EAGT is bigger than BGRT and RAND for all GSL functions. This results show that comparing to BGRT and RAND, EAGT can find higher floating-point inaccuracies with limited resources, especially for large test space (input domain).

TABLE VI
C_v OF MAX. ERROR

Program	EAGT	BGRT	RAND
gsl_sf_airy_ai	0.05%	17.58%	379.29%
gsl_sf_Chi	44.59%	67.03%	178.90%
gsl_sf_Ci	133.32%	489.89%	617.23%
gsl_sf_bessel_J0	77.96%	350.60%	142.64%
gsl_sf_bessel_J1	70.18%	221.44%	171.06%
gsl_sf_bessel_Y0	110.27%	473.18%	308.75%
gsl_sf_bessel_Y1	127.57%	857.70%	443.85%
gsl_sf_eta	74.26%	203.75%	214.93%
gsl_sf_gamma	4.93%	32.37%	1.57%
gsl_sf_legendre_P2	23.58%	13.73%	0.00%
gsl_sf_legendre_P3	17.59%	11.18%	0.00%
gsl_sf_lngamma	7.02%	10.10%	97.67%

B. RQ2

RQ2 asks which method has a better stability. To compare the stability of methods, we use the coefficient of variation [21] as the index, which is defined as the ratio of the standard deviation σ to the mean μ:

$$C_v = \frac{\sigma}{\mu} \tag{14}$$

For every GSL function, we have done 100 trials for each approach. We calculate the value of σ for the 100 results of maximum error and then calculate the ratio of it to the mean of maximum error to achieve the coefficient of variation. The results of C_v are shown in Table VI. The value of C_v can measure the dispersion of the distribution of maximum errors, and the lower value means the lower dispersion. As shown in Table VI, EAGT has lower value of C_v than BGRT for most functions, except for functions $gsl\_sf\_legendre\_P2$ and $gsl\_sf\_legendre\_P3$.

Combining with the validity, we find from Table V that the value of "Ratio on RAND" of BGRT are close to 1 for $gsl\_sf\_legendre\_P2$ and $gsl\_sf\_legendre\_P3$, which means BGRT is equivalent to RAND for these two functions. Meanwhile, the value of "Ratio on RAND" of EAGT are huge for these two functions. Based on this observation, we consider it is meaningless to discuss the stability of an approach if it can not found the high inaccuracy. For example, for the function $gsl\_sf\_legendre\_P2$, the maximum error that BGRT can found is less than 3.0, while the C_v value is small. Another example that can illustrate the instability of BGRT is the function gsl_sf_Ci, for which the value of the mean of maximum error of BGRT is 3.49E+15, but its median of max error is 3.0, which means that more than 50% maximum errors found by BGRT less than 3.0 in 100 trials. The disparity shows that the results of BGRT is of high instability for this function (with C_v value 489.89%).

Answer for RQ2: Based on the analysis above, for 10 of 12 GSL functions (9 of 12 to RAND), EAGT has lower values of C_v than BGRT, which means that EAGT can return the results with lower dispersion. Combining with the validity and the answer for RQ1 that EAGT can find higher inaccuracy than BGRT in 11 of 12 GSL functions, we can conclude that EAGT has better stability to detect high floating-point inaccuracies than BGRT.

C. Threats to Vallidity:

Threats to internal validity are related to errors in our implementation and experiments. We use the mpmath [15] to simulate the real function. We also use the mpmath to avoid the precision-specific operation problem [8]. We consider the operations in mpmath as exact implementations in real arithmetic. We can not ensure, but we try our best to avoid the unexpected error, such as the extra errors in the LSGA approach (see Section III). Threats to external validity are related to the generality of our findings. Currently, our experiment is conducted only on functions with one input variable, which is the basic situation for numerical programs. In the future, we would extend our approach to adopt numerical program with multiple input variables. Threats to construct validity correspond to the suitability of our approach to measure the stability of testing method. We use the coefficient of variation to measure the stability of testing method because it is suitable for the comparison between data sets with different means. We also take into account the validity of the comparison. We consider it is meaningless to discuss the stability of an approach if it can not find the high floating-point inaccuracies.

VII. Discussion

Our approach exploits the global search but not the exhaustive search, so our approach can not guarantee to trigger the highest floating-point inaccuracy. In the future work, we will establish benchmark that includes programs with known maximum floating-point inaccuracy through exhaustive search, and evaluate our approach on this benchmark.

In our experiment, our approach consumes less time than BGRT and RAND in most cases under sequential execution. It is easy to adapt our approach to parallel execution, because comparison only happens after the global and fine search, and the calculation in each $conf$ is independent. In the future work, we will try to implement our approach in parallel, which will significantly reduce the time consumption.

It would be interesting to study the relationship between B_i and C_{fp}. However, it is hard to calculate the value of B_i and the exact value of C_{fp}. In the future work, we will consider approaches that can calculate the exact value of C_{fp} and study the relationship between B_i and C_{fp} to develop possibly more general heuristic rules.

VIII. Conclusion

In this paper, we propose a so-called EAGT approach to efficiently trigger high floating-point inaccuracies in numerical code. Our method uses general heuristic rules that are drawn from error analysis. Under the guidance of those heuristic rules, our method performs an effective global search by approximately computing heuristic values and generate appropriate partitions of the search space. Our experimental results demonstrate that our approach can efficiently find higher floating-point inaccuracies than BGRT and RAND for 11 of 12 real-world GSL functions, and especially effective for programs that have large search space (i.e., input domain). And the results also show that our approach has better stability than BGRT and RAND.

Acknowledgment

This research is supported by the National Key R&D Program of China under Grant No. 2017YFB 1001800, the NSFC under Grant Nos.61379054, 61502015, 61672529, and the Open Project of Shanghai Key Laboratory of Trustworthy Computing under Grant No. 07dz22304201504

References

[1] K. Quinn, "Ever had problems rounding off figures?this stock exchange has," *The Wall Street Journal*, p. 37, 1983.

[2] B. Jakobsen and F. Rosendahl, "The sleipner platform accident," *Structural Engineering International*, vol. 4, no. 3, pp. 190–193, 1994.

[3] S. Putot, E. Goubault, and M. Martel, "Static analysis-based validation of floating-point computations," in *Numerical software with result verification*, pp. 306–313, Springer, 2004.

[4] E. Darulova and V. Kuncak, "Trustworthy numerical computation in scala," in *Acm Sigplan Notices*, vol. 46, pp. 325–344, ACM, 2011.

[5] E. Goubault and S. Putot, "Static analysis of numerical algorithms," in *SAS*, pp. 18–34, Springer, 2006.

[6] W. F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, "Efficient search for inputs causing high floating-point errors.," in *PPOPP*, pp. 43–52, ACM, 2014.

[7] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei, "A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies.," in *ICSE*, pp. 529–539, IEEE Computer Society, 2015.

[8] R. Wang, D. Zou, X. He, Y. Xiong, L. Zhang, and G. Huang, "Detecting and fixing precision-specific operations for measuring floating-point errors," in *FSE*, pp. 619–630, ACM, 2016.

[9] W. Kahan, "Ieee standard 754 for binary floating-point arithmetic," *Lecture Notes on the Status of IEEE*, vol. 754, no. 94720-1776, p. 11, 1996.

[10] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic.," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, 1991.

[11] N. Toronto and J. McCarthy, "Practically accurate floating-point math," *Computing in Science & Engineering*, vol. 16, no. 4, pp. 80–95, 2014.

[12] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 53–64, 2014.

[13] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 1–11, 2015.

[14] F. Benz, A. Hildebrandt, and S. Hack, "A dynamic program analysis to find floating-point accuracy problems," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 453–462, 2012.

[15] F. Johansson *et al.*, *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18)*, December 2013. http://mpmath.org/.

[16] E. T. Barr, T. Vo, V. Le, and Z. Su, "Automatic detection of floating-point exceptions.," in *POPL*, pp. 549–560, ACM, 2013.

[17] S. Boldo and J.-C. Filliâtre, "Formal Verification of Floating-Point Programs.," in *IEEE Symposium on Computer Arithmetic*, pp. 187–194, IEEE Computer Society, 2007.

[18] Z. Fu, Z. Bai, and Z. Su, "Automated backward error analysis for numerical code.," in *OOPSLA*, pp. 639–654, ACM, 2015.

[19] E. Tang, E. T. Barr, X. Li, and Z. Su, "Perturbing numerical calculations for statistical analysis of floating-point program (in)stability.," in *ISSTA*, pp. 131–142, ACM, 2010.

[20] E. Tang, X. Zhang, N. Muller, Z. Chen, and X. Li, "Software numerical instability detection and diagnosis by combining stochastic and infinite-precision testing," *IEEE Transactions on Software Engineering*, 2016.

[21] D. Zwillinger and S. Kokoska, *CRC standard probability and statistics tables and formulae*. Crc Press, 1999.