Hindawi
IET Software
Volume 2023, Article ID 6681267, 16 pages
https://doi.org/10.1049/2023/6681267

Research Article

Detecting Floating-Point Expression Errors Based Improved PSO Algorithm

Hongru Yang, Jinchen Xu, Jiangwei Hao, Zuoyan Zhang, and Bei Zhou

Information Engineering University, No. 62 Science Avenue, High-Tech Zone, Zhengzhou 450001, Henan, China

Correspondence should be addressed to Bei Zhou; 13653970052@163.com

Received 31 May 2023; Revised 7 August 2023; Accepted 16 August 2023; Published 23 October 2023

Academic Editor: Nadeem Sarwar

The use of floating-point numbers inevitably leads to inaccurate results and, in certain cases, significant program failures. Detecting floating-point errors is critical to ensuring that floating-point programs outputs are proper. However, due to the sparsity of floating-point errors, only a limited number of inputs can cause significant floating-point errors, and determining how to detect these inputs and to selecting the appropriate search technique is critical to detecting significant errors. This paper proposes characteristic particle swarm optimization (CPSO) algorithm based on particle swarm optimization (PSO) algorithm. The floating-point expression error detection tool PSOED is implemented, which can detect significant errors in floating-point arithmetic expressions and provide corresponding input. The method presented in this paper is based on two insights: (1) treating floating-point error detection as a search problem and selecting reliable heuristic search strategies to solve the problem; (2) fully utilizing the error distribution laws of expressions and the distribution characteristics of floating-point numbers to guide the search space generation and improve the search efficiency. This paper selects 28 expressions from the FPBench standard set as test cases, uses PSOED to detect the maximum error of the expressions, and compares them to the current dynamic error detection tools S3FP and Herbie. PSOED detects the maximum error 100% better than S3FP, 68% better than Herbie, and 14% equivalent to Herbie. The results of the experiments indicate that PSOED can detect significant floating-point expression errors.

1. Introduction

A large number of floating-point programs are used in key areas such as aerospace, defense, and military, where high reliability is required, so ensuring the accuracy of floating-point program results is critical. However, rounding errors exist between the numerical values used in computer floating-point operations and real values, and the accumulation rounding errors can affect the accuracy of floating-point programs. Currently, most floating-point computing programs use the floating-point precision defined by the IEEE 754 [1] arithmetic standard. When the precision employed does not meet the numerical criteria, the computer will round the value to obtain an estimated value and it is this treatment introduces rounding errors. At the same time, rounding errors will accrue as the operation progresses, and these rounding errors may accumulate to such an extent that they are difficult to ignore, ultimately impacting the correctness of floating-point program calculation outputs. Therefore, error detection techniques are required to detect significant errors in programs and offer input for error triggering, which serves as the foundation for further program optimization. However, it is difficult and inefficient to analyze the error of floating-point program directly. The core numeric operations of a floating-point program can often be abstracted as floating-point expressions, so most methods and tools perform error detection on floating-point expressions.

A lot of research has been done on error detection in floating-point expression, but there are still two areas that can to be improved.

First, prior work does not fully utilized the features of expression error distribution. Consider NMSE problem 336 in FPBench [2], the standard floating-point benchmark. Its expression is $\log(x+1) - \log(x)$. There are 998,768,527 64-bit floating-point numbers in the expression definition field (0.01, 1,000). This is a huge search space, and testing all of the points in it takes time and effort. If we evenly sample the expression on a small scale and calculate the sampling point error, we can find that the input that causes the large error of

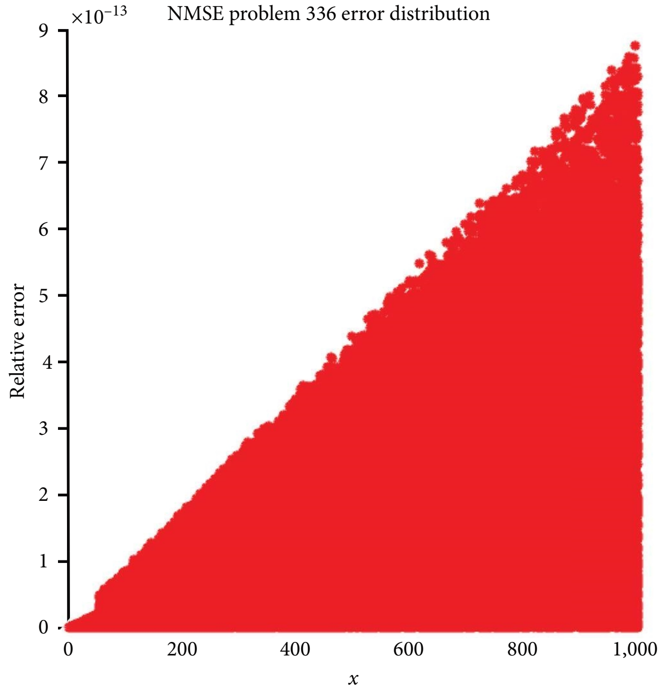


Figure 1: The expression NMSE problem 336 error image.

the expression is mainly concentrated on the right side of the interval, such as the interval (900, 1,000), as shown in Figure 1. By focusing attention on the right side of the interval during the search, the effectiveness of detecting the largest error can be considerably enhanced. With the same number of sample points, the largest errors found using random search in the intervals (0.01, 1,000) and (900, 1,000) were 3.28E-13 and 4.41E-13, respectively. It can be seen that using the error distribution features to guide the search increased the detection impact by 34.5%. The above example shows that it is not necessary to perform a global search on the domain of the floating-point expression every time, but can be searched in some specific intervals to achieve the purpose of reducing the search space and improving the search efficiency. However, current tools fail to take full advantage of this information to aid the search.

Second, existing work which treats floating-point expression error detection as a search problem, using search strategies such as binary search or Monte Carlo Markov to find the maximum error. However, instead of fully exploiting the characteristic floating-point numbers and floating-point errors, the majority of these efforts concentrate on improving the search strategy itself. To produce satisfactory detection results, the algorithm itself should be paired with the properties of the problem. For example, one of the characteristic of floating-point numbers is that they are not evenly distributed along the number line. The closer the floating-point numbers are to the origin, the more concentrated they are. For 64-bit floating-point numbers, the number between (−1, 1) accounts for 49.95% of all 64-bit floating-point numbers. Suppose we use the particle swarm optimization (PSO) algorithm to search for the maximum error. According to the distribution characteristics of floating-point numbers, when the PSO algorithm initializes the population, 50% of the individuals in the population can be initialized outside (−1, 1). Therefore, when improving the existing algorithm and designing the new algorithm, it is necessary to consider how to combine the floating-point number and floating-point error characteristic with the search algorithm for error detection.

To address the aforementioned issues, this paper suggests an error detection method based on the PSO algorithm [3] and implements the error detection tool PSOED. First, the expression is sampled, the error at the sampling point is calculated, and an image is returned to the user. The peak value extraction algorithm is then used to extract the peak points of the image that indicate the potential large error margin. The extracted peak points are then screened for anomalies using the boxplot evaluation index. If outliers can be filtered out, a large error search interval is generated based on the outliers. Finally, the characteristic particle swarm optimization (CPSO) algorithm is applied to generated search interval to perform a maximum error search. This paper makes the following contributions:

(i) A method of constructing detection intervals based on the characteristic of the distribution of expression errors and the distribution of floating-point numbers is proposed, which can effectively assist search algorithm in identifying the significant errors.

(ii) A CPSO algorithm based on the PSO algorithm is proposed, which employs a number of strategies to help the algorithm breaking out of the local optimal solution and can dynamically pick parameters according to the expression.

(iii) PSOED, a floating-point error detection tool, was conceived and built.

(iv) The 28 expressions in FPBench are selected as test cases to validate the effectiveness of the error detection tool.

The rest of the paper is organized as follows. We first introduce the basics of floating-point representation, the definition of the floating-point error, PSO algorithm and boxplot algorithm in Section 2. We then give an overview of our approach in Section 3 and detail our approach in Section 4. We provide the experimental results in Section 5. In Section 6, we summarize related work. We end the paper with conclusion, the limitations of our work, and future work in Section 7.

2. Preliminaries

The basic concepts used in this paper are floating-point number, floating-point error, PSO algorithm, and boxplot. The basic concepts are presented blow.

2.1. Floating-Point Representation. The IEEE 754 standard defines the structure of floating-point numbers and the number of bits in each component of floating-point numbers. Figure 2 shows the number of bits in single and double precision floating point numbers for each component of floating-point numbers.

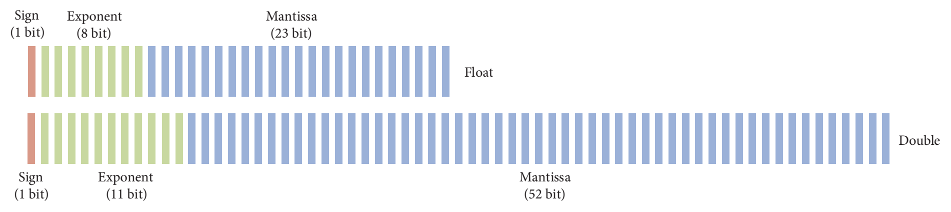


FIGURE 2: Single and double precision floating-point number representation.

2.2. Floating-Point Error. When the floating-point expression is a real number, the error caused by the rounding operation can be defined as the difference between the real value x and the floating-point value $\widehat{x}: x=\widehat{x}+\eta$, where η is the error in representing real numbers as floating-point number.

For floating-point programs $\mathbf{P}: \widehat{y}=\widehat{f(x)}, \widehat{y} \in \mathbb{F}$, errors are introduced and accumulated in each floating-point operation of the program. The accumulation of errors in the operation process leads to the final result containing errors. Relative error ($\text{Error}_{\text{rel}}$) and Absolute error ($\text{Error}_{\text{abs}}$) are two widely used metrics for measuring the magnitude of errors. For the ideal exact result (i.e., the result of a real number operation) $f(x)$ and floating-point program results $\widehat{f(x)}$, the formulae for calculating the relative and absolute errors are as follows:

$$\text{Error}_{\text{rel}}\left(f(x), \widehat{f}(x)\right)=\left|\frac{f(x)-\widehat{f}(x)}{f(x)}\right|, \tag{1}$$

$$\text{Error}_{\text{abs}}\left(f(x), \widehat{f}(x)\right)=\left|f(x)-\widehat{f}(x)\right|. \tag{2}$$

Unit in the last place (ULP) is a unit of measurement of relative error. For a floating-point expression and its corresponding real value: $z=(-1)^S \times d_0 d_1 \ldots d_n \times 2_E$, ULP and ULP error ($\text{Error}_{\text{ULP}}$) can be expressed as follows:

$$\text{ULP}(z)=\frac{\left|d_0 d_1 \ldots d_n-\left(\frac{z}{2^E}\right)\right|}{2^{n-1}}, \tag{3}$$

$$\text{Error}_{\text{ULP}}\left(f(x), \widehat{f}(x)\right)=\left|\frac{f(x)-\widehat{f}(x)}{\text{ULP}(f(x))}\right|. \tag{4}$$

2.3. Particle Swarm Optimization Algorithm. PSO is an intelligent search algorithm based on the group cooperation It is suitable for nonlinear and multipolar problems, and the code is relatively easy to implement with fewer adjustable parameters, so it is widely used in the optimization of parameter rectification. The specific algorithms are:

In the M-dimensional search space, N random particles form a particle swarm, and the position and velocity vectors of each particle are $\vec{x_l}=(x_{i1}, x_{i2}, \ldots, x_{iM})$ and $\vec{v_l}=(v_{i1}, v_{i2}, \ldots, v_{iM})$. Set the position vector $\vec{x_l}=(x_{i1}, x_{i2}, \ldots, x_{iM})$ into the fitness function to calculate the fitness value of the individual particle. The global optimal solution is obtained by judging the magnitude of the fitness value to measure the superiority of the solution and going through successive iterations. Let $p_{\text{besti}}=(p_{i1}, p_{i2}, \ldots, p_{iN})$ be the i-th particle history search to the optimal position. The optimal position searched by the particle swarm history is $g_{\mathbf{besti}}=(g_1, g_2, \ldots, g_N)$. During each iteration, the current velocity and position are updated by the following equation:

$$v_{im}^{k+1}=\omega v_{im}^k+c_1 r_1\left(p_{\text{best}im}-x_{im}^k\right)+c_2 r_2\left(g_{\text{best}} i-x_{im}^k\right), \tag{5}$$

$$x_{im}^{k+1}=x_{im}^k+v_{im}^k, \tag{6}$$

where $i=[1:M]$, c_1 and c_2 are the learning factors, indicating the degree of learning of the particle to its own optimal solution and the population optimal solution; r_1 and r_2 are random numbers on (0,1) and obey a uniform distribution; ω is the inertia weight, indicating the maintenance of the original motion trend of the particle.

2.4. Boxplot. A boxplot is a mathematical algorithm that calculates the dispersion of a data set and is also used to reflect the characteristics of the data distribution. The advantage of the boxplot algorithm is that it relies only on the actual data and does not require the assumption that the data sample set follows a particular distribution shape. On the other hand, the criterion for judging outliers in the boxplot is based on quartiles and interquartile distances. The quartiles are resistant to perturbations, and up to 25% of the data can be arbitrarily far away without perturbing the quartiles significantly, so the outliers do not affect this criterion, which makes the results of anomalous data identification relatively more objective. The principle of the boxplot outlier detection algorithm is shown in the Figure 3.

Q1 is the first quartile of a data set, M is the median, Q3 is the third quartile, and the interquartile range IQR is the difference between the two numbers of Q3 and Q1. If the detected value is less than Q1 − 3IQR or greater than Q3 + 3IQR, the value is considered an extreme outlier; if the detected value is less than Q1 − 1.5IQR or greater than Q3 + 1.5IQR and does not reach the extreme outlier threshold, the value is considered a mild outlier.

3. Overview

The workflow of PSOED, an error detection tool for floating-point expressions, is shown in Figure 4, which consists of an error distribution generation module, an interval generation module and a search module.

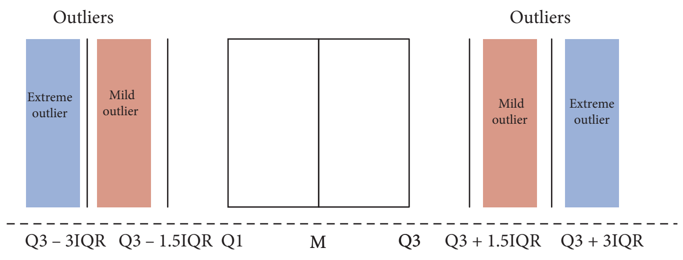


Figure 3: Boxplot evaluation index.

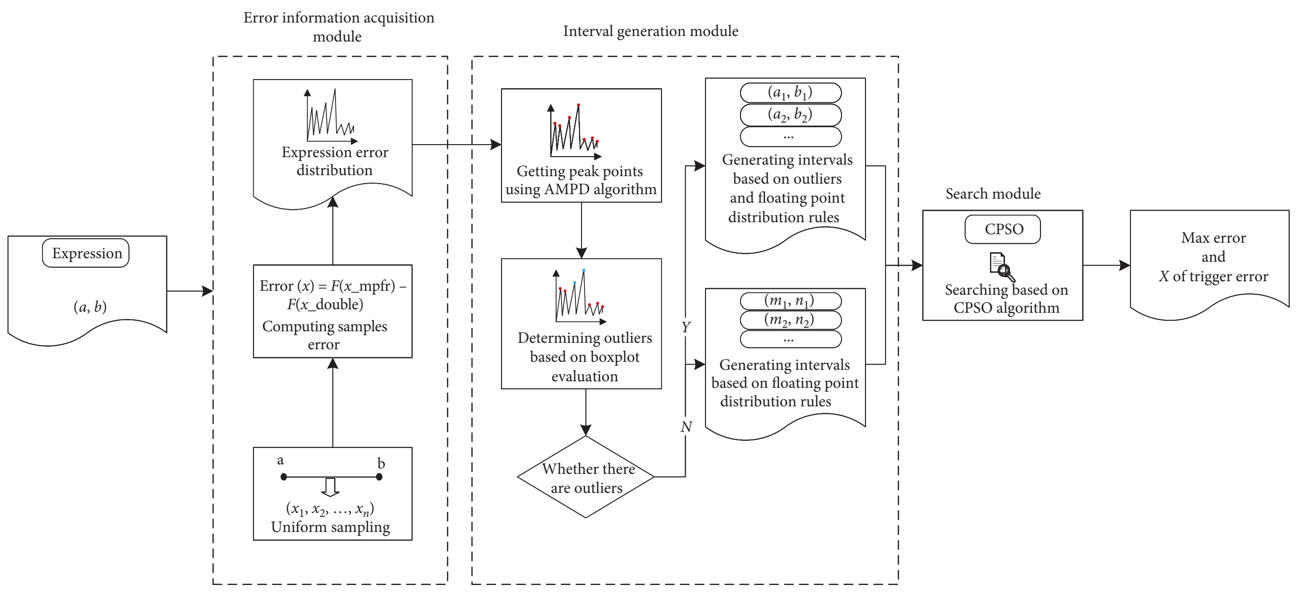

Error information acquisition module
Interval generation module
Expression
(a, b)
Expression error distribution
Error (x) = F(x_mpfr) – F(x_double)
Computing samples error
a
b
(x1, x2, …, xn)
Uniform sampling
Getting peak points using AMPD algorithm
Determining outliers based on boxplot evaluation
Whether there are outliers
Y
N
(a1, b1)
(a2, b2)
…
Generating intervals based on outliers and floating point distribution rules
(m1, n1)
(m2, n2)
…
Generating intervals based on floating point distribution rules
Search module
CPSO
Searching based on CPSO algorithm
Max error and X of trigger error

Figure 4: PSOED workflow diagram.

3.1. Error Distribution Generation Module. The error distribution generation module performs small-scale uniform sampling of the expressions in the respective intervals based on the user input expressions and intervals. We convert the expressions into a low-precision (64-bit) version of the function and a high-precision (128-bit) version of the function based on the MPFR library [4], then calculates the high-precision and low-precision function values according to the sampling points, and the difference between the two is the error result, which is presented to the user in the form of an image.

3.2. Interval Generation Module. Based on error information generated by the error distribution module, the interval generation module will use the AMPD algorithm [5] (see Section 4.2 for details) to extract the error peaks (large error points) in the sampling points, and then calculate the dispersion degree of these error peak points using the boxplot indicator [6] to screen out the outlier error peaks. For any expression, we first generate the interval S_f according to the law of floating-point distribution, and then examine if the error distribution module can extract the outlier peaks from the peak points of the floating-point expression in the sample points. If we can extract the extreme outliers, we apply the bitwise increment/decrement algorithm procedure to produce the big error interval S_e by obtaining the left and right neighborhoods of the extreme outliers.

3.3. Search Module. The search module searches for the maximum error of a floating-point expression in the specified interval using CPSO algorithm (see Section 4.3 for details of the algorithm flow) under the conditions that the user specifies the number of particles and the number of iterations. Finally, the user is given the maximum erro as well as the input that triggered the error.

This tool supports error detection for 28 single-parameter floating-point expressions in FPBench, see Section 5.1 for details of the expressions.

4. Approach

4.1. Obtaining the Error Distribution and Error Peak Points. The error distribution module allows us to obtain the error distribution and peak points of a user input expression over a particular interval. To better describe how this module works, we illustrated its steps.

After obtaining a floating-point expression and the corresponding interval, we first perform a small-scale uniform sampling of the expression to obtain the expression's initial error distribution information. Figures 5–10 depict the error

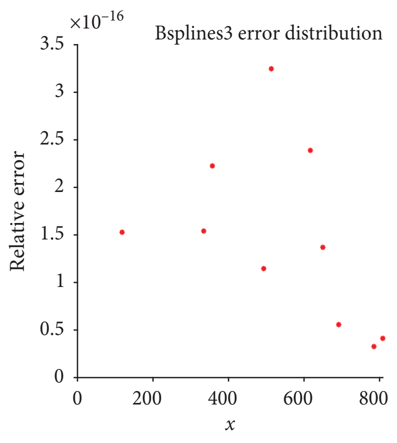


Figure 5: The expression Bsplines3 error image with 10 sampling points.

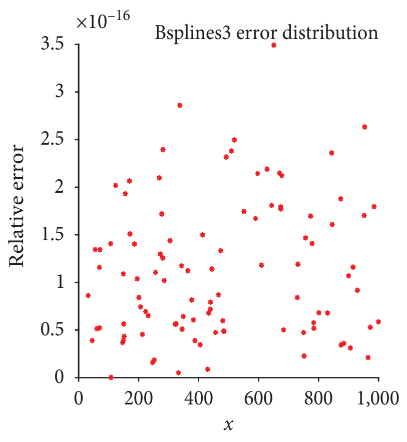


Figure 6: The expression Bsplines3 error image with 100 sampling points.

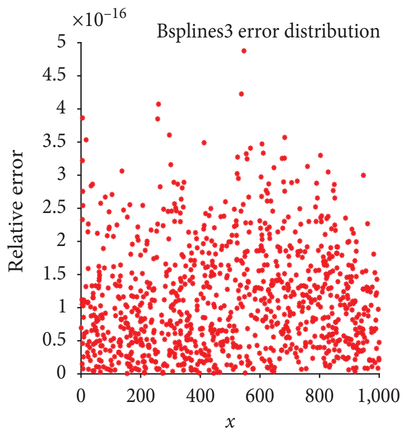


Figure 7: The expression Bsplines3 error image with 1,000 sampling points.

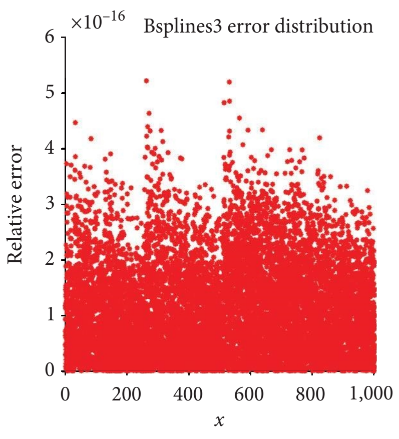


Figure 8: The expression Bsplines3 error image with 10,000 sampling points.

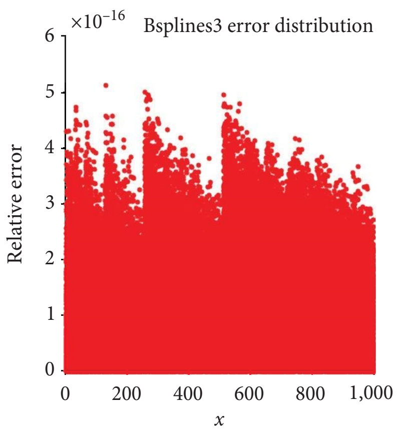


Figure 9: The expression Bsplines3 error image with 100,000 sampling points.

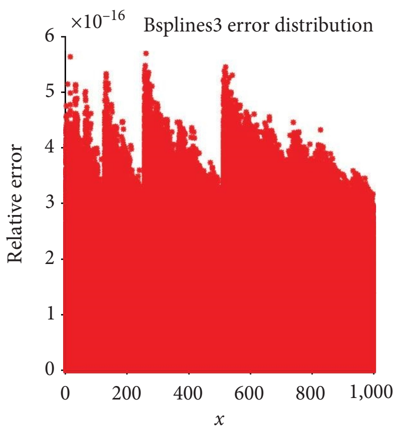


Figure 10: The expression Bsplines3 error image with 1,000,000 sampling points.

```
Require: expression $f(x)$, $[a, b]$
1: Us ⇐ Uniform_sample(a, b, 100,000)
2: for $x_i$ of Us do
3:     temp_error ⇐$f_{MPFR}(x_i) - f_{double}(x_i)$
4:     error_list.append(temp_error)
5: end for
6: peak_list ⇐ AMPD(error_list)
7: return peak_list, error_list
```

Algorithm 1: Extracting peak points based on AMPD algorithm.

plots for sampling points of 10, 100, 1,000, 10,000, and 1,000,000 on the interval (0, 1,000) using the Bsplines3 expression as an example, respectively.

As shown in Figure 5, when the sampling point is 100,000, the error distribution of the Bsplines3 expression is basically the same as when the sampling point is 1,000,000. Despite the fact that the more sampling points there are, the more accurate the error distribution is. We set the default sampling point to 100,000 for the performance and accuracy reasons, and all expressions adhere to this standard. Of course, the number of sampling points can be adjusted by the user.

After obtaining the error distribution of the expression, we focus on the image's local maxima, which are referred to as error peak points. In order to obtain the peak points on the image more accurately and quickly, we introduce the multiscale-based automatic peak detection (AMPD) algorithm in the field of signal processing. This algorithm was proposed by Scholkmann et al. [5]. It is a noise and signal peak detection algorithm based on local maxima. In this paper, we can think of the error image as a noise (or signal) image and apply the AMPD algorithm to get the noise (or signal) peak points, i.e., the desired error peak points. In this paper, we will use it directly without going into the details of the AMPD algorithm. The error image peak point extraction based on the AMPD algorithm is described in Algotithm 1.

As shown in Figure 11, after the AMPD algorithm extracts the peak points, we label the peak points on the error image to reflect the detection effect: the black points on the image are the error peak points found by the error distribution module.

4.2. Generating Large Error Intervals Based on Error Peaks. After the processing of the error distribution module, we get the peak points, the interval generation module will process the peak points to generate the interval, the workflow is divided into two steps. First, the peak points are filtered and sorted to generate outlier peak points. Then, large error intervals are generated based on the presence or absence of outlier peaks.

Since the peak points selected by the AMPD algorithm are local maxima, but the local maxima can be large or small, and the error of two peak points can be several orders of magnitude different, the first step we need to do in this module is to eliminate the peak points with small errors

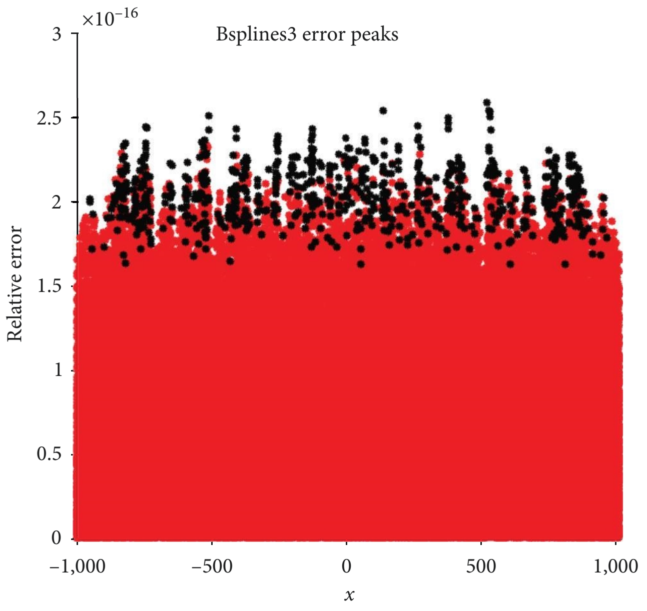


Figure 11: Error peaks image.

and keep only the peak points with relatively large errors. To eliminate the smaller peaks, we need to use an evaluation index to evaluate the degree of anomaly of each peak in the set of peaks.

The error data samples are discrete data that have the characteristics of strong randomness and do not obey specific distribution laws. Among the mature algorithms commonly used for data outlier detection, the 3σ rule or z-score method based on normal distribution have the prerequisite of assuming that the data obey normal distribution, but the error data samples do not obey normal distribution, and the criteria for determining the outliers of the above methods are based on calculating the mean and standard deviation of the full set of samples, and the mean and standard deviation have very little interference resistance, so the outliers themselves will have a large impact on the performance of the algorithm. Therefore, the outliers themselves have a significant impact on the performance of the algorithm. Taking the 3σ rule as an example, σ represents the standard deviation and μ represents the mean, the normal interval of values is considered to be $(\mu - 3\sigma, \mu + 3\sigma)$, and the probability that the values are distributed in this interval is 99.7%, that is, the number of outliers obtained by this algorithm is not more than 0.3% of the total number of sample sets. Therefore, this algorithm is obviously not the optimal solution. The boxplot algorithm is a good solution to this problem.

The standard given in the boxplot is used to evaluate the degree of dispersion of each peak point. Extreme outliers and mild outliers are considered as outlier peak points. We can obtain two results: $\mathscr{A}$, there is no outlier peak (normal value) and $\mathscr{B}$, there is an outlier peak (extreme outlier). These two results correspond to two characteristics of the error distribution: uniform distribution and nonuniform distribution. In both cases, we must first generate the interval Sf according to the floating-point distribution law. The existence of outlier

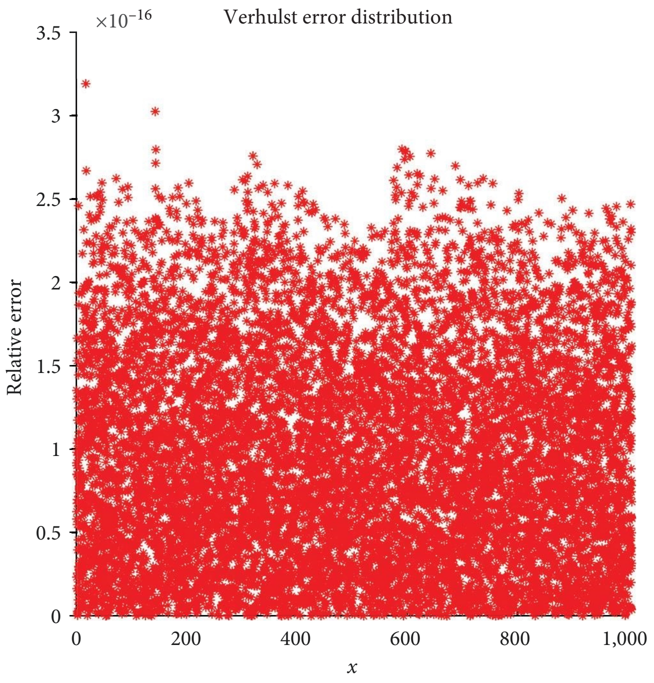


Figure 12: The expression verhulst error image.

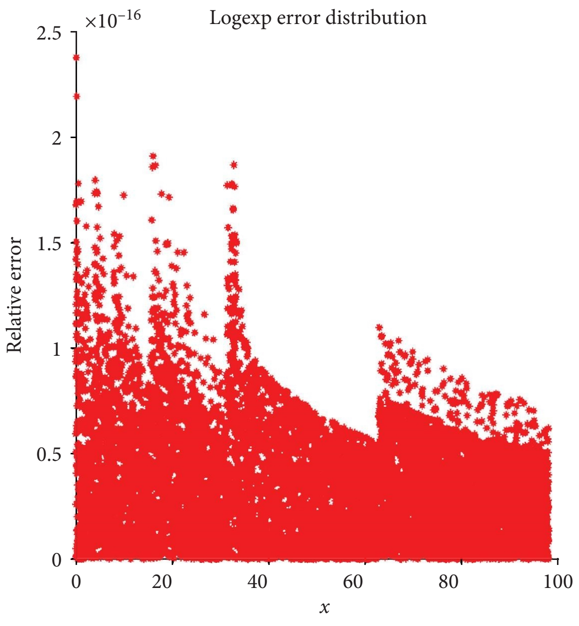


Figure 13: The expression logexp error image.

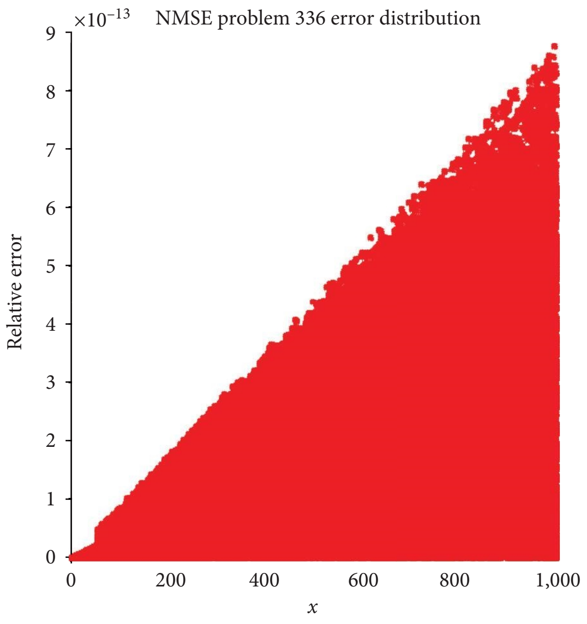


Figure 14: The expression NMSE problem 336 error image.

peaks reflects whether the error distribution is uniform. In case $\mathscr{A}$, the absence of outlier peaks indicates that the difference between each peak is small, which means that the error of the expression is uniformly distributed in the given interval. The error distribution of the expression verhulst in the interval (0, 1,000) is shown in Figure 12.

At this point, it can be seen that the error information is not very meaningful here because the error is uniformly distributed and there are many peaks. For such expressions, no special interval generation is done, and the interval Sf is used as the search band.

Floating-point numbers are not evenly distributed along the number line. The closer you get to the origin, the denser the distribution. For floating-point numbers of type double, the numbers between (−1,1) make up 49.95% of all floating-point numbers, or about half. For this distribution property, two cases are considered when initializing the particle population: above (−1,1) and outside (−1,1).

For case $\mathscr{B}$, the presence of outlier peaks indicates that the errors are prominent. For example, the errors of the expressopm logexp, NMSE problem 341, and exp1x_log are shown in Figures 13–15, respectively. Their errors do not have a uniform distribution.

For expressions with nonuniform error distribution, a series of outlier peaks can be obtained by calculating according to the boxplot method. The peaks set Sp and the corresponding error set E_p can be obtained by sorting. According to the points in the outlier peak point set S_p, the size of the left and right neighborhoods can be determined according to the bit increment/decrement search algorithm. Taking p_0 as an example, to determine the size of the left neighborhood, point p_0 is decremented by bit, and the error size of the inputs is calculated after each decrement. If it is less than e_1, the iteration is stopped. The determination of the right neighborhood is similar to that of the left neighborhood. Note that the last peak point is discarded. Two factors are taken into account. One is that there is no reference threshold when searching by bit for the last peak point. According to this search method, the left and right neighborhood intervals $\overbrace{S_I}$ and $\underbrace{S_I}$ are generated, respectively, and the final large error interval $S_I = \overbrace{S_I} \bigcup \overbrace{S_I}$. It is worth noting that

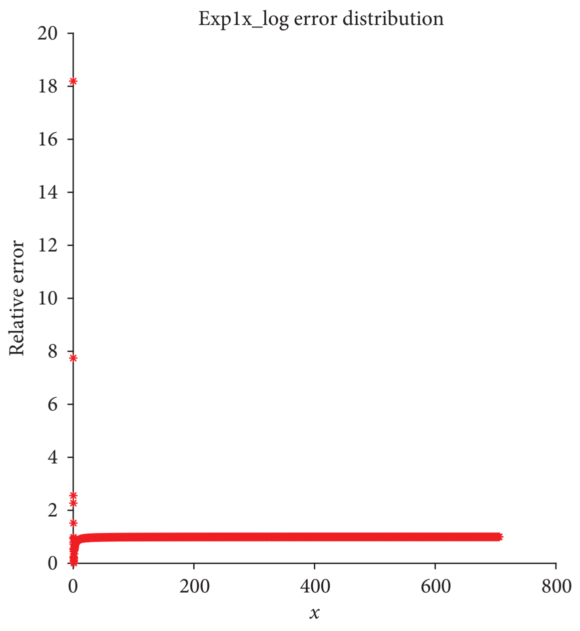


FIGURE 15: The expression exp1x_log error image.

if there are two intervals I_m and I_n, and $I_m \bigcap I_n$, the two need to be merged into $I' = I_m \bigcup I_n$. Here is an example:

Through calculation, a peak point set S_p and corresponding error set can be obtained as follows:

$$S_p : \begin{cases} p_0 \rightarrow 0.2312 \\ p_1 \rightarrow 0.2634 \\ p_2 \rightarrow 53.4564 \\ p_3 \rightarrow 348.3242 \\ p_4 \rightarrow 72.9586 \end{cases} E_p : \begin{cases} e_0 \rightarrow 7.32e^{-14} \\ e_1 \rightarrow 7.11e^{-14} \\ e_2 \rightarrow 7.33e^{-15} \\ e_3 \rightarrow 3.41e^{-15} \\ e_4 \rightarrow 1.17e^{-15} \end{cases}. \quad (7)$$

Using a bit search for each point in S_p, you can obtain the left interval $\overbrace{S_E}$ and the right interval $\underbrace{S_E}$.

```
Require: peak_list, error_list
Ensure: $S_I$
1: if $peak\_list.length \neq 0$ then
2:   for $index < peak\_list.length$ do
3:     if $index = = peak\_list, length - 1$ then
4:       break
5:     else
6:       left, right $\Leftarrow$ peak_list[i]
7:       while $Cal\_error(left) > error\_list[index + 1]$ do
8:         left $\Leftarrow$ left-0x1
9:       end while
10:        while $Cal\_error(right) > error\_list[index + 1]$ do
11:          right $\Leftarrow$ right + 0x1
12:        end while
13:        $S_I$[index] $\Leftarrow$ [left, right]
14:     end if
15:     index $\Leftarrow$ index + 1
16:   end for
17:   $S_E \Leftarrow$ merge_intervals($S_I$)
18: else
19:   $S_F \Leftarrow$ float_distribution(a, b)
20: end if
21: if $S_E$ then
22:   $S_I \Leftarrow S_E \bigcup S_F$
23: else
24:   $S_I \Leftarrow S_F$
25: end if
26: return $S_I$
```

ALGORITHM 2: Generating large error intervals based on outlier peaks.

$$\overbrace{S_e} : \begin{cases} I_0 \rightarrow (0.1324, 0.2312) \\ I_1 \rightarrow (0.1906, 0.2634) \\ I_2 \rightarrow (49.2189, 53.4563) \\ I_3 \rightarrow (321.3423, 348.3242) \end{cases} \underbrace{S_e} : \begin{cases} I_0 \rightarrow (0.2312, 1.3172) \\ I_1 \rightarrow (0.2634, 1.7503) \\ I_2 \rightarrow (53.4564, 60.1231) \\ I_3 \rightarrow (348.3242, 362.2131). \end{cases} \quad (8)$$

Combine $\overbrace{S_E}$ and $\underbrace{S_E}$ to obtain the final large error interval set S_E.

$$S_E : \begin{cases} I_0 \rightarrow (0.1324, 1.7503) \\ I_1 \rightarrow (49.2189, 60.4564) \\ I_2 \rightarrow (321.3423, 348.3242) \end{cases}. \quad (9)$$

These intervals will divide the number of particles equally, assuming that the resulting set of large error intervals S_E contains n intervals, and we calculate the number of CPSO initialization particles on each interval using Equation (6).

Finally, S_E and S_F form S_I as the search interval for the CPSO algorithm. The overall process of the algorithm is shown in Algorithm 2.

4.3. Searching Using CPSO Algorithm. The main disadvantages of the standard PSO algorithm are: (1) it is easy to fall into local optimal solutions; (2) it is highly dependent on the problem.

In this paper, the standard PSO algorithm is improved, and the new algorithm is call CPSO algorithm, and the specific improvement strategies and reasons are as follows:

4.3.1. The Perturbation Parameter τ Is Added. In the standard PSO algorithm, the particle will only move to the position with a better fitness value than itself, which is easy to fall into the local optimal solution. To help the particles jump out of the local optimal solution, we borrow the idea from the Annealing algorithm [7] and set a perturbation parameter τ, which represents the probability that the particle will move to the position with a fitness value lower than its own. In order not to affect the convergence of the algorithm, this parameter decrease linearly with the increase of the number of iterations. In the early iterations, the parameter increases the global search ability of the particle, and in the late iterations, it does not affect the normal convergence of the algorithm. As shown in Equation (10).

$$\tau = \tau_{\max} - (\tau_{\max} - \tau_{\min}) \times \frac{\text{Iter}_{\max} - \text{Iter}_{\text{now}}}{\text{Iter}_{\max}}. \qquad (10)$$

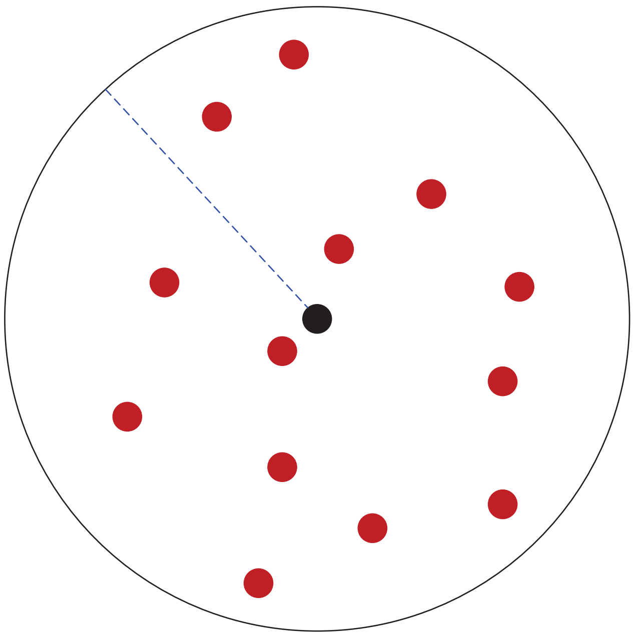


Figure 16: Associated particle schematic.

4.3.2. Each Particle Will Be Followed by Associated. Given the sparsity of errors in floating-point expressions, even small changes can produce large error results, so we introduce the concept of associated particles, as shown in Figure 16. The associated particles are all points in the space with a particle at the center of the circle, let's call it the central particle, of radius x. After the associated particles are captured, we compare the fitness value of the central particle with that of the associated particles, one by one. If the fitness value of an associated particle is greater than that of the central particle, the associated particle will replace the central particle and inherit a number of attributes from the central particle. Each time the central particle moves, it creates a new batch of companion particles. Of course, we will not select all of a particle's associated particles. The initial value designed by this algorithm is 1,000, which can be adjusted by the user.

4.3.3. When the Velocity or Position of the Particle Crosses the Boundary, the "Collision-Bounce" Strategy Is Used. Due to the interval limitation of the expression, the situation of position and velocity overshooting should also be considered in the particle motion process. In the case of overshooting, this paper adopts the collection-bounce strategy. If the particle position overshoot size is equal to the interval length, the particle is allowed to return; otherwise, the particle position is regenerated and the velocity is the global optimum. The calculation method is shown in Equations (11) and (12) as follows:

$$x_{im}^{k} = \begin{cases} x_{\min} + \left|x_{im}^{k} - x_{\min}\right|, \text{ if } x_{m}^{k} < x_{\min} \wedge \left|x_{m}^{k} - x_{\min}\right| < (x_{\max} - x_{\min}) \\ x_{\max} - \left|x_{im}^{k} - x_{\max}\right|, \text{ if } x_{m}^{k} > x_{\max} \wedge \left|x_{m}^{k} - x_{\min}\right| < (x_{\max} - x_{\min}) \end{cases}, \qquad (11)$$

$$v_{im}^{k} = \begin{cases} v_{\min} + \left|v_{im}^{k} - v_{\min}\right|, \text{ if } v_{m}^{k} < v_{\min} \wedge \left|v_{m}^{k} - v_{\min}\right| < (v_{\max} - v_{\min}) \\ v_{\max} - \left|v_{im}^{k} - v_{\max}\right|, \text{ if } v_{m}^{k} > v_{\max} \wedge \left|v_{m}^{k} - v_{\max}\right| < (v_{\max} - v_{\min}) \end{cases}. \qquad (12)$$

4.3.4. Parameters Vary with the Expression. In the standard PSO algorithm, we need to set a value for each given parameter. The idea of this paper is that the parameters should depend on the problem, while keeping the idea of the algorithm unchanged. For different expressions, different parameters should be selected by the algorithm, and the specific values of the parameters most applicable to the expressions are shown in the experimental part of the fourth, which will not be described here.

4.3.5. Fitness Function. This paper provides three fitness functions–the relative error, ULP error, and bit error of the expression corresponding to the particle under its value-in order to be able to compare the tool proposed in this paper

```
Require: expression $f(x)$, $S_I$
1: Particles ⇐ population_initialization($S_I$)
2: gBest ⇐ getGlobalBest(particles)
3: iter ⇐ 0
4: while $iter < maxIteration$ do
5:     for $particle\ of\ particles$ do
6:         vUpdate(particle)
7:         pUpdate(particle)
8:         updateLocalBest(particle)
9:         UpdateGlobalBest(particle)
10:        associatedParticles ⇐ geneAssociatedParticles()
11:    end for
12:    for $aParticle\ of\ associatedParticles$ do
13:        if $particle.error > gBest\_error$ then
14:            gBest ⇐ aParticle
15:        end if
16:    end for
17:    iter ⇐ iter + 1
18: end while
19: MaxError ⇐ gBest.error
20: x ⇐ gBest.x
21: return MaxError, x
```

Algorithm 3: MaxError search based on CPSO.

with other tools. Since different tools use different types of error, this paper's tool needs to be compared with other tools, and different tools use different types of error, too. The following equations provide these three fitness functions.

$$\text{Fitness}_{\text{rel}}\left(f(x),\hat{f}(x)\right) = \left|\frac{f(x)-\hat{f}(x)}{f(x)}\right|, \quad (13)$$

$$\text{Fitness}_{\text{ULP}}\left(f(x),\hat{f}(x)\right) = \left|\frac{f(x)-\hat{f}(x)}{\text{ULP}(f(x))}\right|, \quad (14)$$

$$\text{Fitness}_{\text{bits}}\left(f(x),\hat{f}(x)\right) = \lceil \log_2 \text{Fitness}_{\text{ULP}} \rceil. \quad (15)$$

After processing the interval generation module in the second step of the tool, we now have a specific search interval. The next work is to use the CPSO algorithm to find the maximum error on the search interval.

As shown in Algorithm 3, first, the values of each parameter are determined by the expression. Then, the particle population is initialized to the search interval generated in the second step, the fitness value of each particle is calculated, and the global optimum is obtained. Meanwhile, the velocity direction of all particles is set to the global optimum. Lines 6–16 of the algorithm represent an iterative process. As shown in the algorithm, in an iterative process, the velocity and position of each particle in the particle swarm must be updated. In the updating process, it is necessary to judge whether the velocity and position of each particle are out of bounds. After updating the particle's velocity and position, the particle's associated particles are generated. According to the fitness of the associated particles, it is judged whether the central particle and the global optimum need to be updated. At this point, an iteration process is completed. After the maximum number of iterations, the maximum error found by the algorithm and the point that caused the error are stored in the global optimum, and the response value is obtained by the global optimum. Finally, the maximum error corresponding to the input expression is returned to the user and the error input is triggered.

5. Evaluation

First, the values of parameters in CPSO algorithm were determined by experiments. Second, random search, PSO, and CPSO were used, respectively, to detect and compare the errors of FPBench test case expressions to verify the effectiveness of CPSO algorithm. Finally, PSOED was compared with S3FP and Herbie to verify the effectiveness of PSOED.

5.1. Test Cases and Experimental Environment. The body code of PSOED was implemented in C++. The body code was approximately 2,000 lines. We selected 28 single-parameter expressions from FPBench benchmarks that did not contain loops and judgments. Instead of using the default range provided by FPBench, we used a large custom range. Information about the expressions is shown in Table 1.

The hardware and software environment required for the experiment is shown in Table 2.

5.2. CPSO Parameter Determination. In this paper, the parameters of the CPSO algorithm are determined from an experimental point of view. For example, the expression exp1 with the interval (0, 708) is used.

When other parameters are fixed, the test starts with parameters $c_1 = 1.0$ and $c_2 = 1.0$. Figure 17 shows the average error of the expression searched by CPSO algorithm when c_1 and c_2 change dynamically. This is paper performs 12 sets of tests for each expression in Table 3, removes the maximum and minimum values of the experimental results, and calculates the average value of the remaining data, which is the obtained results as shown in Equation (16). In Figure 17, the coordinate representation of the bars (c_1, c_2) and the height represent the average maximum error. When c_1 and c_2 are different, the bars in statistical graph use different colors. As shown in Figure 17 when $c_1 = 1.4$ and $c_2 = 2.0$, the CPSO algorithm achieves the best effect on the expression exp1x. Figure 18 shows that when ω takes different values and there is a graph with $\omega = 0.3$, the average error obtained by CPSO is the largest.

$$\text{MaxError}_{\text{ave}} = \frac{\sum_{1}^{\text{Times}} \text{error} - \text{error}_{\max} - \text{error}_{\min}}{\text{Times} - 2}. \quad (16)$$

c_1, c_2, and ω corresponding to different expressions are shown in Table 4.

Table 1: Expressions.

NO.	FPBench	Expression	$\mathcal{D}$
1	Bsplines3	$-x*x*x/6$	$[-1000, 1000]$
2	Exp1x	$(\exp(x)-1)/x$	$[0.01, 708]$
3	Exp1x_log	$(\exp(x)-1)/\log(\exp(x))$	$[0.01, 708]$
4	Intro_example	$t/(t+1)$	$[0, 1000]$
5	Logexp	$\log(\exp(x)+1)$	$[0.01, 1000]$
6	NMSE example 3.1	$\mathrm{sqrt}(x+1)-\mathrm{sqrt}(x)$	$[0, 1000]$
7	NMSE example 3.4	$(1-\cos(x))/\sin(x)$	$[0.01, 1000]$
8	NMSE example 3.5	$\mathrm{atan}(x+1)-\mathrm{atan}(x)$	$[-1000, 1000]$
9	NMSE example 3.6	$1/\mathrm{sqrt}(x)-1/\mathrm{sqrt}(1+x)$	$[0, 1000]$
10	NMSE example 3.7	$\exp(x)-1$	$[0.01, 10]$
11	NMSE example 3.8	$(x+1)*\log(x+1)-x*\log(x)-1$	$[0.01, 1000]$
12	NMSE example 3.9	$1/x-1/\tan(x)$	$[0.01, 1000]$
13	NMSE problem 331	$1/(x+1)-1/x$	$[-1000, 1000]$
14	NMSE problem 333	$1/(1+x)-2/x+1/(x-1)$	$[-1000, 1000]$
15	NMSE problem 336	$\log(x+1)/\log(x)$	$[0.01, 1000]$
16	NMSE problem 337	$\exp(x)-2+\exp(-x)$	$[0.01, 100]$
17	NMSE problem 341	$(1-\cos(x))/(x*x)$	$[0.01, 1000]$
18	NMSE problem 344	$\mathrm{sqrt}((\exp(2*x)-1)/(\exp(x)-1))$	$[0.01, 304]$
19	NMSE problem 345	$(x-\sin(x))/(x-\tan(x))$	$[0.01, 1000]$
20	NMSE section 3.11	$\exp(x)/(\exp(x)-1)$	$[0.01, 708]$
21	predatorPrey	$(4*x*x)/(1+x/1.11*x/1.11)$	$[-1000, 1000]$
22	Sine	$x-(1.0/6.0)*x*x*x$ $+(1.0/120.0)*x*x*x*x*x$ $-(1.0/5040.0)*x*x*x*x*x*x*x$	$[-1000, 1000]$
23	Sineorder3	$0.954929658551372*x$ $-0.12900613773279798*x*x*x$	$[-1000, 1000]$
24	sqroot	$0.0625*x*x*x-0.0390625*x*x*x*x$	$[-1000, 1000]$
25	Sqrt_add	$1/(\mathrm{sqrt}(x+1)+\mathrm{sqrt}(x))$	$[0, 1000]$
26	Test05_nonlin1, r4	$(x-1)/(x*x-1)$	$[-1000, 1000]$
27	Test05_nonlin1, test2	$1.0/(1+x)$	$[0, 1000]$
28	verhulst	$(4*x)/(1+x/1.11)$	$[-1000, 1000]$

Table 2: Experimental environment.

Name	Version
CPU	Intel Xeon E5-2630 v4
Clock	2.10 GHZ
Memory	64 GB
Cache sizes	64 KB L1, 1,024 KB L2, 2,816 KB L3
OS	Ubuntu 20.04.5 LTS
Compiler	GCC v9.4.0
Compilation options	-lmpfr -lm

5.3. Comparison of Random, PSO, and CPSO Search Algorithm. In this section, we compare the performance of PSO algorithm and CPSO algorithm, and the search effect of random search, PSO algorithm and CPSO algorithm.

In terms of performance, this paper compares the performance of PSO algorithm with that of CPSO algorithm. The reason why it is not compared with random search is that both PSO and CPSO have a series of judgment and iteration strategies, while random search does not have any judgment process, so the performance must be better than the two. In terms of error, we use relative error to describe the search effect of different algorithms. In order to reflect the search ability of PSO and CPSO, the random generation strategy was adopted for the particle population of PSO and CPSO. After obtaining the error results, in order to show on a statistical chart that the maximum error detected by different expressions is not necessarily in the same order of magnitude, we used a logarithmic scale with a base of 10 for the error results, as shown in the following formula.

$$\mathrm{Error}_{\mathrm{display}} = \log_{10}(\mathrm{Error}_{\mathrm{relative}}). \quad (17)$$

As shown in Figure 19, the performance of the CPSO algorithm is better than that of standard PSO for 18 expressions (64%). Compared with PSO, CPSO has added a series of judgment and processing strategies, such as regeneration of particle velocity and position after transgression. After

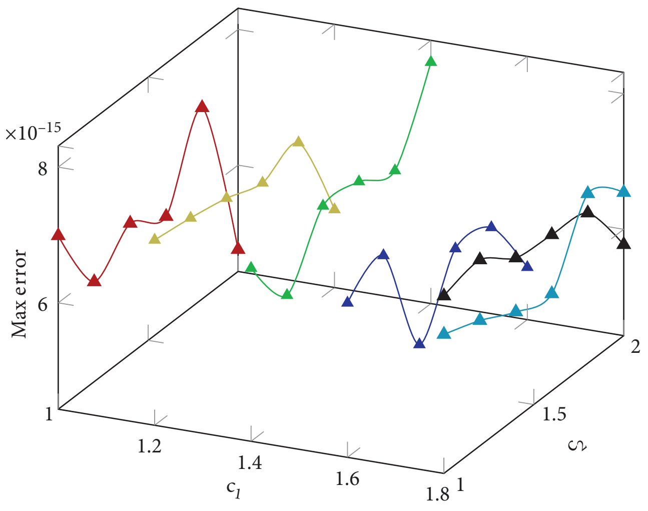


Figure 17: c_1 and c_2 of the CPSO for the expression exp1x.

Table 3: Init population according to floating-point distribution.

	Init population interval	
Range	S_f	Particlenumber
$b \le -1$	$[a, b]$	M
$a \le -1 < b \le 1$	$[a, -1], [-1, b]$	$M/2, M/2$
$a \le -1 < b < 1$	$[a, -1], [-1, 1], [1, b]$	$M/4, M/2, M/4$
$-1 < a < b < 1$	$[a, b]$	M
$-1 < a < 1 \le b$	$[a, 1], [1, b]$	$M/2, M/2$
$1 \le a < 1$	$[a, b]$	M

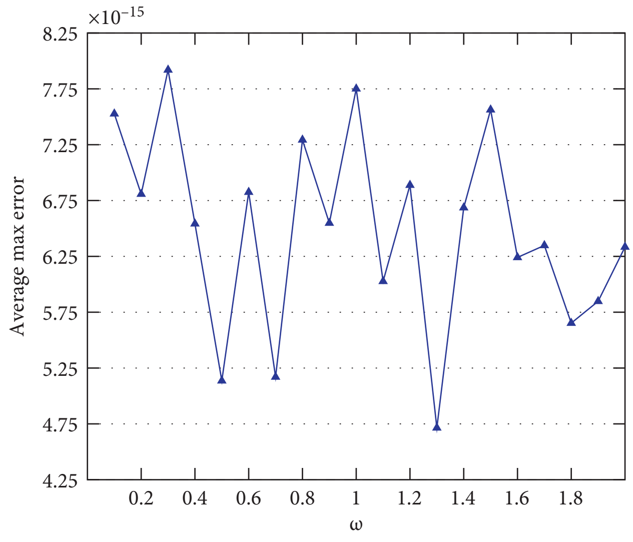


Figure 18: ω of the CPSO algorithm for the expression exp1x.

adding a series of judgment and processing strategies, such performance results are acceptable.

As shown in Figure 20, the search performance of CPSO algorithm is better than that of random search algorithm in 28 (100%) expression and better than that of PSO algorithm in 17 (61%) expressions.

Table 4: Parameter values for different expressions.

No	FPBench	c_1	c_2	ω
1	Bsplines3	1.4	1.6	0.9
2	Exp1x	1.4	2.0	0.3
3	Exp1x_log	1.4	1.8	1.5
4	Intro_example	1.2	1.8	1.9
5	Logexp	1.4	1.0	1.1
6	NMSE example 3.1	1.6	1.6	1.8
7	NMSE example 3.4	1.6	1.4	1.4
8	NMSE example 3.5	1.8	1.2	1.7
9	NMSE example 3.6	1.0	2.0	1.7
10	NMSE example 3.7	1.6	2.0	1.9
11	NMSE example 3.8	1.8	2.0	2.0
12	NMSE example 3.9	1.6	1.4	1.7
13	NMSE problem 331	1.2	1.4	0.9
14	NMSE problem 333	1.0	1.4	1.4
15	NMSE problem 336	1.0	1.0	0.7
16	NMSE problem 337	1.8	1.8	0.7
17	NMSE problem 341	2.0	2.0	0.4
18	NMSE problem 344	1.2	1.4	1.5
19	NMSE problem 345	2.0	1.2	1.8
20	NMSE section 3.11	1.8	1.8	0.6
21	predatorPrey	2.0	1.0	0.8
22	Sine	1.6	1.8	0.3
23	Sineorder3	1.8	1.6	0.5
24	sqroot	1.8	2.0	0.7
25	Sqrt_add	1.6	2.0	1.6
26	Test05_nonlin1,r4	1.0	1.8	1.0
27	Test05_nonlin1, test2	1.8	1.0	1.4
28	verhulst	1.4	1.8	0.9

5.4. Comparison of PSOED, S3FP, and Herbie. For the 28 expressions selected for the experiment, the PSOED test results are shown in Table 5.

Since the PSOED tool proposed in this paper is a dynamic detection tool, it needs to be compared with the existing dynamic detection tools. We selected S3FP and Herbie.

S3FP is a binary guided (BGRT) error detection tool, and the test result is a relative error. Since the test time can be set for S3FP, we set the same run time for S3FP and PSOED to ensure the fairness of the test, and compare the maximum errors found by the two tools. There are 14 expressions that S3FP does not support and are represented by no answer (NA) in this paper. The test results are shown in Figure 21.

As shown in the figure above, the maximum error obtained by PSOED is greater than that obtained by S3FP for 28 expressions in the experimental case for the same run time.

Herbie is an error detection and precision adjustment tool that uses random sampling and heuristic search to estimate and locate expression errors. Herbie's error representation is

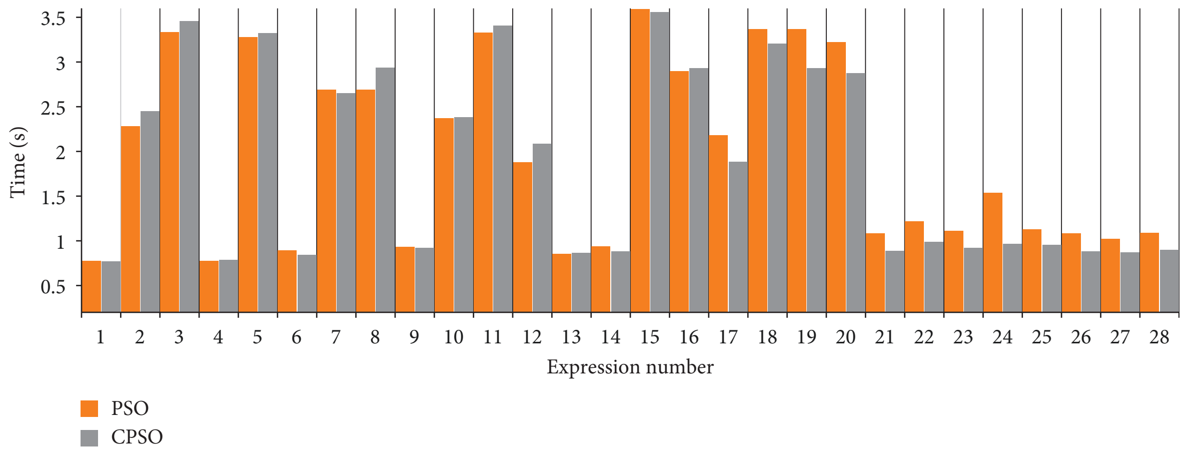


FIGURE 19: Performance comparison between PSO and CPSO.

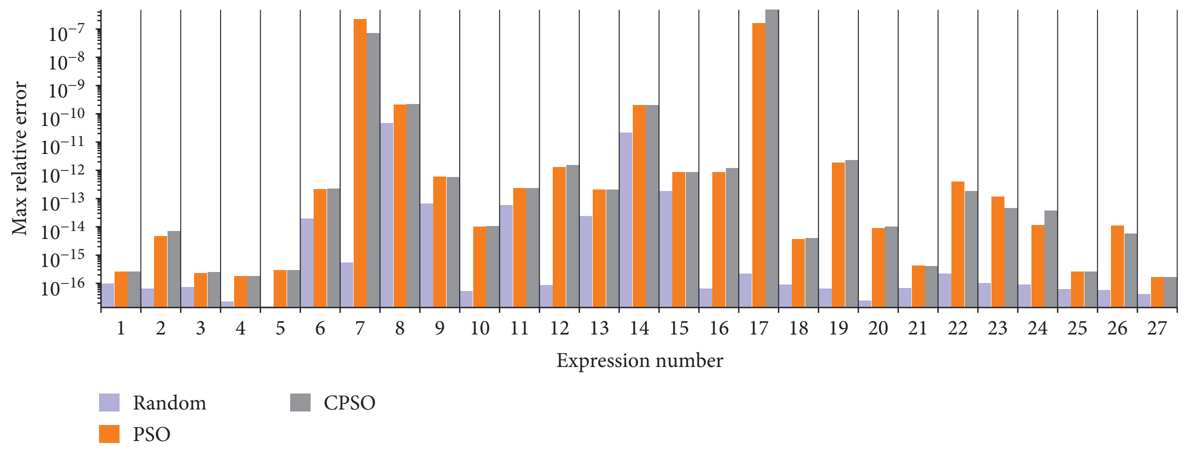


FIGURE 20: Comparison of search effect random, CPSO, and CPSO.

$\text{ReError}_{\text{bits}}$. The two tools use different error measures. For ease of comparison, Equation (16) was used to convert the ULP error results measured by PSOED to $\text{ReError}_{\text{bits}}$. The sampling points set by Herbie was 8,000, and we the number of particles of PSOED was set to 8,000.

$$\text{ReError}_{\text{bits}} = \begin{cases} 0 & \text{ReError}_{\text{ULP}} \leq 0.5 \\ 1 & 0.5 < \text{ReError}_{\text{ULP}} \leq 1 \\ \lceil \log_2 (\text{ReError}_{\text{ULP}}) \rceil & \text{ReError}_{\text{ULP}} > 1 \end{cases}. \tag{18}$$

In terms of detection effect, as shown in Figure 22, among the 28 expressions used in the experiment, 19 (68%) expressions with PSOED had a better maximum detection effect than Herbie, 4 (14%) expressions with the same detection effect, and 5 (18%) expressions with PSOED had a lower maximum error than Herbie.

In terms of tool performance, as shown in Figure 23, only 11(40%) of the PSOED expressions analyzed had a lower detection time than Herbie. For the expressions that cost more time than Herbie, it is because there are many large error intervals generated in the second step of the tool, and PSOED performs multiple searches, resulting in high time cost. As shown in Figure 23, the expression NMSE example 38 has a much larger time overhead than Herbie, we analyze the error distribution of this expression and find that there are many peaks in the error image of this expression, so PSOED generates 174 large error intervals when processing this expression, which means that PSOED has to search at least 175 times, and the time cost is obvious.

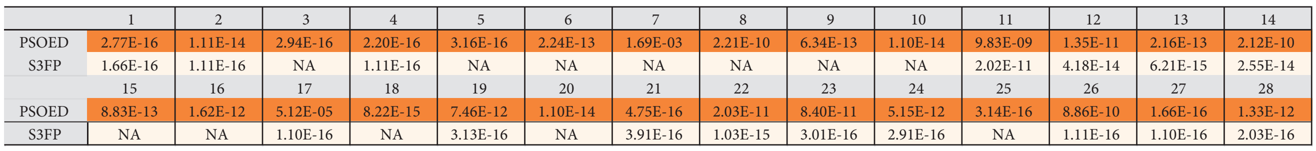
6. Related Work

6.1. Static Tools. Static analysis tools are based on the source code analysis and verify the accuracy of numerical operations by collecting key information from programs. Currently, there are many mature static analysis tools, such as Gappa [8], Daisy [9], FPTaylor [10], Fluctuat [11], and Rosa [12]. Static analysis tools are suitable for the specific areas (such as embedded systems), because programs in these fields do not have complex and dynamic data structures. Then, they rely on the program source code, and the detected errors are larger than the actual errors, so there are false positives.

6.2. Dynamic Tools. In contrast to dynamic analysis tools and methods, Chiang et al. [13] evaluated the accuracy of static

Table 5: PSOED test results.

No.	FPBench	PSOED Max relative error	Max ULP error	Max bits error
1	Bsplines3	2.77E-16	1.00E + 00	1.00
2	Exp1x	1.11E-14	2.00E + 00	5.67
3	Exp1x_log	2.94E-16	5.00E + 01	1.58
4	Intro_example	2.2E-16	1.00E + 00	1.00
5	Logexp	3.16E-16	1.00E + 00	1.00
6	NMSE example 3.1	2.24E-13	1.36E + 03	10.43
7	NMSE example 3.4	1.69E-3	1.68E + 09	30.98
8	NMSE example 3.5	2.21E-10	1.05E + 06	20.00
9	NMSE example 3.6	6.34E-13	3.58E + 03	11.79
10	NMSE example 3.7	1.10E-14	6.40E + 01	6.02
11	NMSE example 3.8	9.83E-09	3.66E + 07	26.04
12	NMSE example 3.9	1.35E-11	4.57E + 05	15.63
13	NMSE problem 331	2.16E-13	1.27E + 03	10.30
14	NMSE problem 333	2.12E-10	1.04E + 06	20.00
15	NMSE problem 336	8.83E-13	4.59E + 03	12.15
16	NMSE problem 337	1.62E-12	1.23E + 04	13.58
17	NMSE problem 341	5.12E-05	1.85E + 09	30.64
18	NMSE problem 344	8.22E-15	5.10E + 01	5.73
19	NMSE problem 345	7.47E-12	6.88E + 04	16.05
20	NMSE section 3.11	1.10E-14	7.70E + 01	6.29
21	predatorPrey	4.75E-16	3.00E + 00	1.58
22	Sine	2.03E-11	1.27E + 05	18.56
23	Sineorder3	8.40E-11	8.99E + 04	15.80
24	sqroot	5.15E-12	3.41E + 05	12.85
25	Sqrt_add	3.14E-16	2.00E + 00	1.58
26	Test05_nonlin1, r4	8.86E-10	4.43E + 06	21.65
27	Test05_nonlin1, test2	1.66E-16	1.00E + 00	1.00
28	verhulst	1.33E-12	1.60E + 04	13.90



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
PSOED	2.77E-16	1.11E-14	2.94E-16	2.20E-16	3.16E-16	2.24E-13	1.69E-03	2.21E-10	6.34E-13	1.10E-14	9.83E-09	1.35E-11	2.16E-13	2.12E-10
S3FP	1.66E-16	1.11E-16	NA	1.11E-16	NA	NA	NA	NA	NA	NA	2.02E-11	4.18E-14	6.21E-15	2.55E-14
	15	16	17	18	19	20	21	22	23	24	25	26	27	28
PSOED	8.83E-13	1.62E-12	5.12E-05	8.22E-15	7.46E-12	1.10E-14	4.75E-16	2.03E-11	8.40E-11	5.15E-12	3.14E-16	8.86E-10	1.66E-16	1.33E-12
S3FP	NA	NA	1.10E-16	NA	3.13E-16	NA	3.91E-16	1.03E-15	3.01E-16	2.91E-16	NA	1.11E-16	1.10E-16	2.03E-16

Figure 21: Comparison of detection effect between PSOED and S3FP. NA, no answer.

analysis and SMT-based methods, and proposed their advantages and disadvantages for the first time. They designed a binary guided random search algorithm (BGRT), and implemented an S3FP tool to assist parallel developers in practical fault analysis. Zou et al. [14] and Zou et al. [15] conducted an empirical analysis and found that the size of the exponential bit and the composition of the mantissa bit can have a significant impact on accuracy, i.e., floating point numbers have only one index within a cell, which can lead to significant inaccuracy, while a large portion of the mantissa bit can lead to significant inaccuracy. They designed a new genetic algorithm, the locally sensitive genetic algorithm (LSGA) [14]. This algorithm improves the genetic algorithm [16] and combines the original algorithm with floating-point number composition. The basic idea is to evolve exponential bits to hit the cells with significant errors and at the same time randomly generate mantissa bits. The fitness function value of the algorithm is calculated by the FPDebug [17] tool. Zou et al. [14] and Zou et al. [15] later proposed to detect floating-point errors based on atomic conditions. Their view is that atomic conditions are an important factor in floating-point errors in atomic operations, and a floating-point error is represented as: $\varepsilon_{\text{out}} = \varepsilon_{\text{in}}\tau + \mu$ and τ are the atomic condition, τ refers to the error introduced by atomic operations. The IEEE754 standard and the GNU C [18] reference manual guarantee that the latter is very small because atomic operations are carefully implemented and maintained by experts, so only atomic conditions need to be considered. Atomic conditions guide the search process for significant errors, and the tool Atomu [15] is implemented. Panchekha et al. [19] proposed Herbie, which

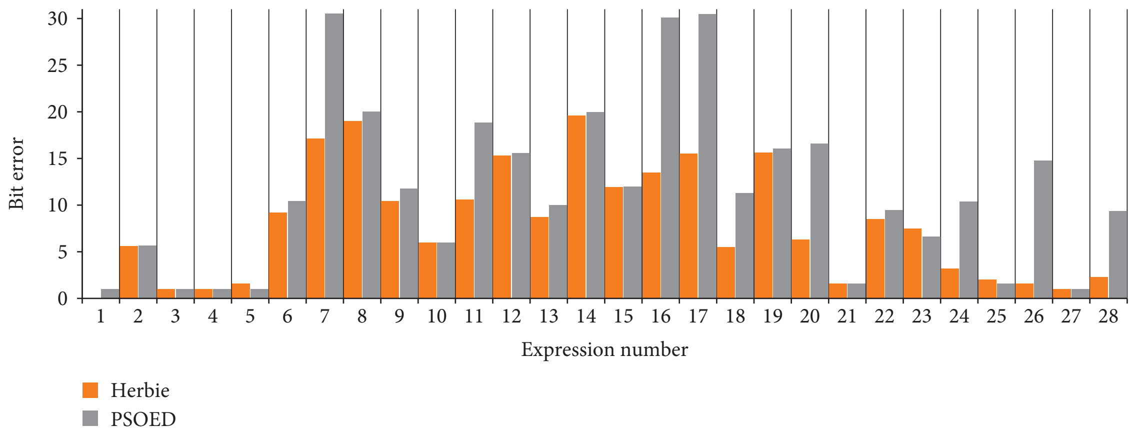


Figure 22: Comparison of detection effect between PSOED and Herbie.

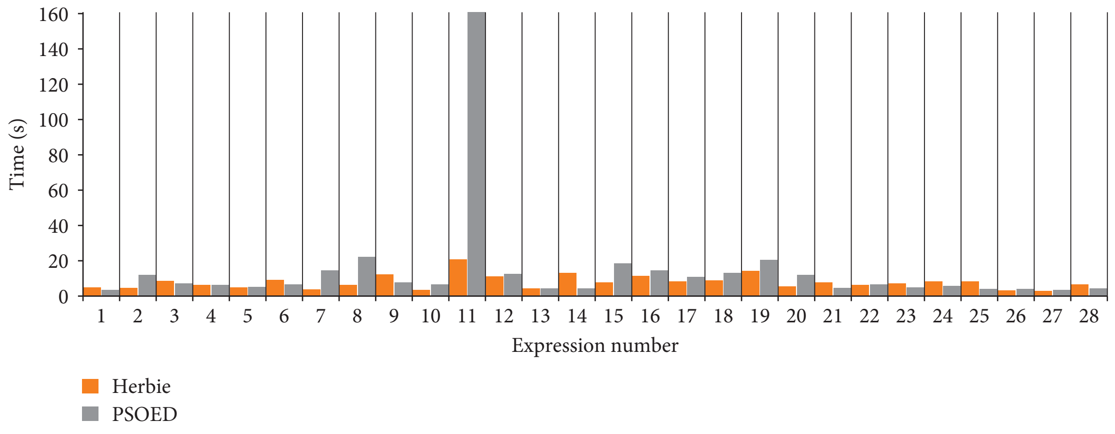


Figure 23: Comparison of performance between PSOED and Herbie.

is an error detection and accuracy adjustment tool that uses a random search method to locate areas with large errors, and then improves accuracy by rewriting the expressions in those areas. Guo and Rubio-González [20] believe that the loss of accuracy and the elimination of errors are the key factors leading to large errors in a program, so they are defined by the formulas. Using the symbol execution technology in software testing, the two conditions are injected into the program, and then the input that triggers the above two large error conditions, namely the high-error induced input, is checked. The FPGen tool is implemented.

7. Conclusion and Future Work

Due to the significant impact of floating-point errors on floating-point programs, floating-point detection is a fundamental work for precision optimization or mixed precision, and many researchers have conducted research on it. Various search-based techniques or tools for detecting the maximum error of floating-point programs or expressions have achieved good results, but it is still a very challenging problem. In this paper, the error distribution law of floating-point expression and the distribution characteristics of floating-point number are fully considered, and the CPSO algorithm is used to detect the maximum error of floating-point expression. These work are all very important, organically combined to form a good tool PSOED. According to the experimental results, the CPSO algorithm shows superiority over the random search algorithm and the standard PSO algorithm. The comparison with S3FP and Herbie tools proves the effectiveness and practicality of the tool.

It has to be admitted that PSOED tools have a lot of potential for optimization. For one thing, we are trying to support error detection for multiparameter expressions with loops and judgments. On the other hand, we need to optimize the performance of PSOED. Too many large error intervals lead to excessive processing time. In the future, we plan to use some strategy to reduce the size of large error intervals while paralleling the search process.

In order to identify significant errors in floating-point expressions, our research aims to enhance the PSO algorithm and apply it to the field of floating-point computing. There are also many excellent optimization algorithms that are used in the different areas. For example, using butterfly optimization algorithm (BOA) [21] to solve the problem of the setting of design parameters of designed ADRC controller, using gray-wolf optimization (GWO) [22] algorithm to improve the performances of SMC and STSMC [23] and using social spider optimization (SSO) algorithm to tune the parameters of proportional–derivative (PD) versions of both IT2FLC and T1FLC [24]. However, our work cannot directly be compared to the aforementioned work due to the diverse research areas and methodologies employed, however these works have strong reference relevance. We will, of course, continue to research the techniques you proposed in the future with the goal of coming up with a more effective solution to the problem of floating-point expression error detection. On the other hand, in order to verify the practicality of PSOED tools, we will test a large number of examples in the field of engineering and computer science in the future.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Authors' Contributions

Hongru Yang and Jinchen Xu contributed equally to this work.

References

[1] D. Zuras and M. Cowlishaw, "IEEE standard for floating-point arithmetic," in *IEEE Std 754™-2008 (Revision of IEEE Std 754-1985)*, pp. 1–58, IEEE, New York, NY, 29 August 2008.

[2] N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock, "Toward a standard benchmark format and suite for floating-point analysis," in *Numerical Software Verification. NSV 2016*, S. Bogomolov, M. Martel, and P. Prabhakar, Eds., vol. 10152 of *Lecture Notes in Computer Science*, pp. 63–77, Springer, Cham, 2017.

[3] R. Poli, J. Kennedy, and T. Blackwell, "Particle swarm optimization," *Swarm Intelligence*, vol. 1, pp. 33–57, 2007.

[4] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: a multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, pp. 13–es, 2007.

[5] F. Scholkmann, J. Boss, and M. Wolf, "An efficient algorithm for automatic peak detection in noisy periodic and quasi-periodic signals," *Algorithms*, vol. 5, no. 4, pp. 588–603, 2012.

[6] Y. Liu, "Box plots: use and interpretation," *Transfusion*, vol. 48, no. 11, pp. 2279-2280, 2008.

[7] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[8] F. de Dinechin, C. Lauter, and G. Melquiond, "Certifying the floating-point implementation of an elementary function using Gappa," *IEEE Transactions on Computers*, vol. 60, no. 2, pp. 242–253, 2011.

[9] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian, "Daisy-framework for analysis and optimization of numerical programs (tool paper)," in *Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2018*, D. Beyer and M. Huisman, Eds., vol. 10805 of *Lecture Notes in Computer Science*, pp. 270–287, Springer, Cham, 2018.

[10] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions," *ACM Transactions on Programming Languages and Systems*, vol. 41, no. 1, pp. 1–39, 2018.

[11] E. Goubault, "Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT," in *Static Analysis. SAS 2013*, F. Logozzo and M. Fähndrich, Eds., vol. 7935 of *Lecture Notes in Computer Science*, pp. 1–3, Springer, Berlin, Heidelberg, 2013.

[12] E. Darulova and V. Kunčak, "Towards a compiler for reals," *ACM Transactions on Programming Languages and Systems*, vol. 39, no. 2, pp. 1–28, 2017.

[13] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, "Efficient search for inputs causing high floating-point errors," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pp. 43–52, Association for Computing Machinery, 2014.

[14] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei, "A genetic algorithm for detecting significant floating-point inaccuracies," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering 1*, pp. 529–539, IEEE, Florence, Italy, 2015.

[15] D. Zou, M. Zeng, Y. Xiong, Z. Fu, L. Zhang, and Z. Su, "Detecting floating-point errors via atomic conditions," *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 1–27, 2019.

[16] D. E. Goldberg, "Genetic Algorithms in Search Optimization and Machine Learning," 1988.

[17] 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.

[18] S. Loosemore, R. Stallman, A. Oram, and R. McGrath, "The GNU C Library Reference Manual," 2001.

[19] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–11, Association for Computing Machinery, 2015.

[20] H. Guo and C. Rubio-González, "Efficient generation of error-inducing floating-point inputs via symbolic execution," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1261–1272, Association for Computing Machinery, 2020.

[21] S. Arora and S. Singh, "Butterfly optimization algorithm: a novel approach for global optimization," *Soft Computing*, vol. 23, pp. 715–734, 2019.

[22] S. Mirjalili, S. M. Mirjalili, and A. Lewis, "Grey wolf optimizer," *Advances in Engineering Software*, vol. 69, pp. 46–61, 2014.

[23] A. A. Al-Qassar, A. Abdulkareem, A. F. Hasan et al., "Grey-wolf optimization better enhances the dynamic performance of roll motion for tail-sitter vtol aircraft guided and controlled by STSMC," 2021.

[24] A. J. Humaidi, H. T. Najem, A. Q. Al-Dujaili, D. A. Pereira, I. K. Ibraheem, and A. T. Azar, "Social spider optimization algorithm for tuning parameters in PD-like interval type-2 fuzzy logic controller applied to a parallel robot," *Measurement and Control*, vol. 54, no. 3-4, pp. 303–323, 2021.