Error Detection for Floating-Point Program via Branch and Bound Method

Kai Song[1], Xia Zeng[1], Min Tang[2]

[1]Shanghai Key Laboratory of Trustworthy Computing,East China Normal University,Shanghai, China

[2]School of Mathematics and Computing Science ,Guilin University of Electrical Technology ,Guilin, China

e-mail: kaisong@ecnu.cn

Keywords:floating-point; error detection; program rewriting; branch and bound;GNU Scientific Library

Abstract. It is well-known that writing an error-free floating-point program is very difficult. Thus, detecting unacceptable errors of a floating-point program is important. In this paper, we develop a system named SpaceAED. The main function of this system is to automatically detect unacceptable errors of a floating-point program written in C programming language. The key insight of this work is to use interval arithmetic in conjunction with branch and bound technique. The implementation of SpaceAED is to rewrite a floating-point program to one that can run on interval arithmetic, and then use branch and bound technique to find all inputs that can trigger unacceptable errors. We choose a great many of functions in GNU Scientific Library (GSL) to test SpaceAED, including matrix computations and evaluation of special functions etc. Numerical results show that SpaceAED is available for accurately detecting unacceptable error-triggering inputs of numerical functions.

Introduction

On June 4, 1996, Ariane 5 rocket, launched by European Space Agency, ended in failure because of an error that converts data from a 64-bit floating point number to a 16-bit signed integer value to overflow[1]. On February 3, 2010, Toyota recalled vehicles because of anti-lock brake software [2]. Numeric program, which manipulates floating-point arithmetic, plays a critical role in many fields of national defense, transportation, finance. Clearly, nowadays our people increasingly rely on numeric program. Floating-point numbers are the finite precision encoding of real numbers, the result of their operations are not exactly representable but rounded[3]. Rounding errors, if it manages to accumulate sufficiently, may probably destroy a numeric result[4, 15]. C language is the most widely used programming language in industry such as in aerospace engineering, due to a combination of characteristics such as code portability, efficiency, low runtime system resource demand and so on[5]. Therefore, research on error analysis for numeric program written in C language has important scientific value and practicalsignificance.

Theoretical analysis on floating-point arithmetic has been extensively studied. Jean-Michel Muller systematically presents basic concepts of floating-point arithmetic including formats, exceptions, rounding modes etc.[6]. Ramon E. Moore presents basics of interval arithmetic, which is the most common used method to keep track of and analyze rounding errors arising from each floating-point operation [4]. However, little work has been carried on error analysis of floating-point code. In this paper, we consider a problem that how to detect all of inputs triggering unacceptable error specified by users for a given numeric program, which is a challenging problem.

Based on interval arithmetic, we propose an algorithm using branch and bound method[7] to efficiently detect all the unacceptable error-triggering inputs of a given floating-point program. To this end, we rewrite a numeric program to one that can be run on interval arithmetic. Then we run the rewritten program with the given inputs. When it terminates, the rewritten program will report all unacceptable error-triggering inputs.

Our approach, which we have called SpaceAED, automatically detects all unacceptable error-triggering inputs. Program that run on an arbitrary pair of those inputs will certainly produce an unacceptable error. When SpaceAED finds all the unacceptable error-triggering inputs, developers are able to exactly use a program without triggering unacceptable error.

To realize SpaceAED, we first use Flex and Bison[8] to generate abstract syntax tree(AST) by building C language syntax analyzer. Then we rewrite numeric program written in C language to the form that can run on Boost Library[9] by traversing AST. Finally we finish the module of detecting all the unacceptable error-triggering inputs by using branch and bound method[10,11].

Our contributions are as follows.

- A practical method for detecting unacceptable error based on branch and bound method.
- A system that can automatically detect all the unacceptable error-triggering inputs and its evaluation on the GNU Scientific Library(GSL).

The paper is organized as follows. Section 2 gives a program drawn from GSL to clarify our problem. Section 3 introduces how to implement program rewriting and how to complete error detection via branch and bound method. In section 4, we take three examples to demonstrate howSpaceAED works out in practice and show the results of experiment. Some concluding remarks are made in Section 5.

```
typedef struct{
double dat[2];
}gsl_complex;
gsl_complexgsl_complex_exp (gsl_complex a){
   /* z=exp(a) */
double rho = exp (GSL_REAL (a));
double theta = GSL_IMAG (a);
gsl_complex z;
 GSL_SET_COMPLEX (&z,
        rho * cos (theta), rho * sin (theta));
 return z;
}
```

Figure 1. GSL's implementation of gsl_complex_exp.

ILLUSTRATIVE EXAMPLE

In order to clarify our problem, we use a function, **gsl_complex_exp**(gsl_complex z), drawn from the GSL complex functions. And it returns the complex exponential of the complex number z. Fig. 1gives the GSL's implement-
ation of gsl_complex_exp.

We declare a gsl_complex variable a as input and take a.dat[0]=1.53 and a.dat[1]=2.15. Then we declare another gsl_complex variable b as input and take b.dat[0]=1.531 and b.dat[1]=2.151. Here we use$||f(b)-f(a)||_2/||b-a||_2$to obtain relative error, where*f*means gsl_complex_exp and $||||_2$denotes 2-norm. According to the previous formula, the relative error is 1212.9. However, in the case that a.dat[0]=
10.82,a.dat[1]=12.26,b.dat[0]=10.821 and b.dat[1]=12.261, the relative error is 1.06512e+007.

Based on the analysis of the two results above, it is obvious that for a given program, the relative error at some point is very small while that at another point is very large enough to probably destroy a numeric result. Therefore, it is important and necessary to detect all inputs of gsl_complex_
exp that can trigger unacceptable error specified by users.

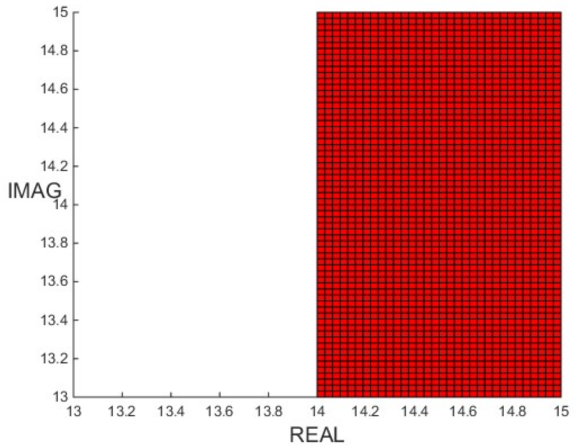

Figure 2. A region formed byall unacceptable error-triggering inputs.

Let dat[0] and dat[1] be bounded by interval [13,15]. Run on this function, SpaceAED reports that all the inputs on dat[0] and dat[1], which trigger unacceptable error. The shadow area in Fig. 2 is formed by all the unacceptable error-triggering inputs on dat[0] and dat[1] and the small shadow rectangle represents one unacceptable error-triggering inputs region.

APPROACH

Since SpaceAED analyzes programs not functions, the architecture of SpaceAED, which is given by Fig. 3, describes two main phases. Phase one automatically rewrites a numeric program written in C language into one that can be run on Boost library, which is mature, well-tested, well-maintained andprovides support for interval arithmetic[9]. During phase two, SpaceAED runs the rewritten program with inputs on interval and reports all the inputs that trigger unacceptable error when it terminates. The key challenge in its implementation is how to efficiently detect all those inputs of one program, whose input are usually multiple parameters.

We first describe the implementation of program rewriting of our error detection tool, SpaceAED (see Section 3.1) and how we detect all unacceptable error-triggering inputs for a given floating-point program (see Section 3.2).

A. *Program Rewriting*

In order to transform a numeric program written in C language into a form that can be run on Boost library, we design a transformer that takes a numeric program written in C language and rewrites its **float** and **double** types into **intv_float** and **intv_double** respectively, both of which are defined in Boost library.Besides rewriting types, it also rewrites floating-point relation expression. For instance, *x*>= *y* is transformed to *x*.*lower*() >= *y*.*upper*() and *x*<= 0.5 is transformed to *x*.*upper*() <= 0.5. In addition to previous rewriting rules, it also has another rewriting rule $R(C[e])=C[R[e]]$ in global context, where R denotes our transformer, *e* means an expression or declaration of floating-point, and C stands for a numeric program context.

Based on the rewriting rules above and characteristics of C programming language, we will implement program rewriting based on abstract syntax tree, which is often used in program analysis and program transformation systems[13].

1) *Abstract Syntax Tree Generation*

Abstract syntax tree is a data structure widely used in compliers and often serves as a tree representation of a program. Generating AST takes three steps: lexical analysis, syntax analysis and semantic analysis.

In order to generate abstract syntax tree, we exploit Flex and Bison, which are open source tools for building programs that handle structural inputs. Fig. 4 depicts the process of abstract syntax tree generation.

a) Lexical analysis

We group characters into lexical units or tokens. It is implemented by using Flex to write lexical analysis program lex.l, compiling it into .c file, and executing .c file to decompose the source code into tokens.

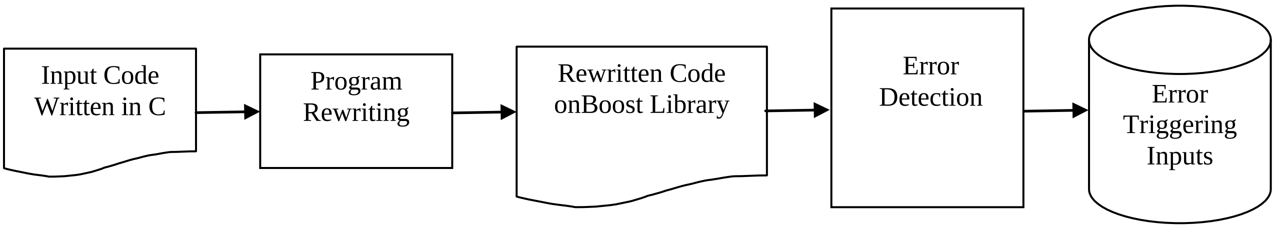

Figure 3. The architecture of SpaceAED.

b) Syntax analysis

We group tokens into syntactical units, is implemented by using Bison to write syntax analysis program parser.y, compiling it to generate .c file and .h file, and executing the .c file and .h file to get a parse tree representation of the program. However, the parse tree is not completed and will be refined by semantic analysis.

c) Semantic analysis

We analyze the parse tree for context-sensitive in-formation concerning variables and other objects, which is stored in a symbol table and produce a completed syntax tree containing all the information of the structure of the program.

In this paper, we generate the abstract syntax tree based on object-oriented principle, every node in which represents a syntax structure, such as identifier, expression, statement, declaration, compound statement, function declaration, function body.

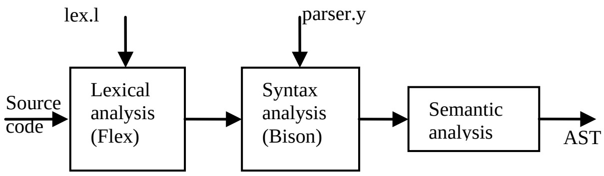

Figure 4. Process of abstract syntax tree generation.

2) Code Transformation

Through the previous process, abstract syntax tree and symbol table have already been generated. Then we will implement code transformation by traversing the syntax tree. Here we will exploit recursive call technique to traverse syntax tree. The below is our algorithm **CTAST**(code transformation via AST) for transforming code.

Algorithm CTAST. This algorithm takes a rewriting rule list L and root of a given numeric program p, then returns root of rewritten numeric program *new_p*.

(1) IF L is not empty THEN
    (a) Set $CR \leftarrow L[1]$ and remove the first item from L;
    (b) RewriteFunc(p)
        IF(p is NULL) THEN
            RETURN;
        ELSE
            IF(semantic of p conform to CR)
                1.*new_p*:= rewrite(p);
                2.RETURN *new_p*;
        ElSE
                For(from left to right traversing each subNode SN of p)
                    RewriteFunc(SN);

END For

END IF

END IF

END IF

Remark. In step(a), *CR* means current rewriting rule. In step(b), subroutine *rewrite* is an concrete function for how to transform code based on *CR* in details.

B. *Error Detection*

For the problem that how to detect all unacceptable error-triggering inputs of a given floating-point pro-gram, the biggest challenge is that how to efficiently solve it in the case that the program has *n*-tuple input and *m*-tuple output.

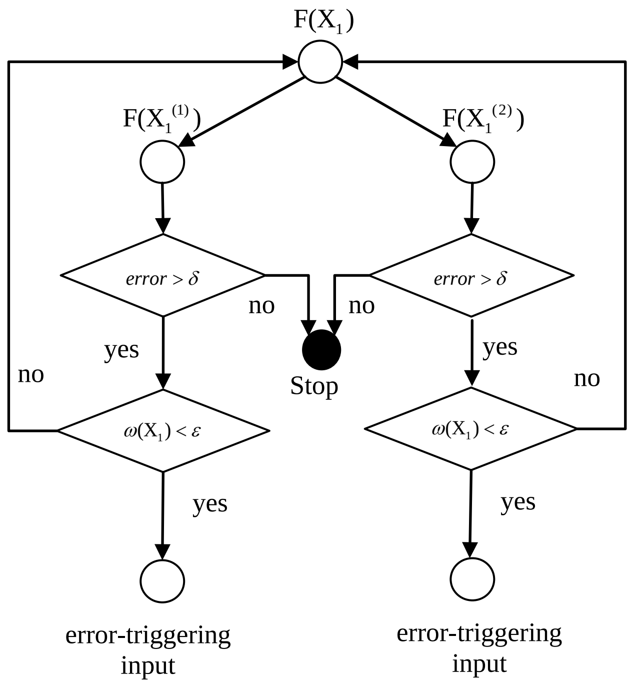

Figure 5. Process of unacceptable error-triggering inputs detection for one program of one parameter.

For a program of n-tuple input, the naive way to error detection is to equally split every dimension of *n*-tuple input interval into subdivisions whose width are less than a given tolerance. And we compute the error triggered by the program for each possible combination, then return all the subdivisions' combination that cause unacceptable error.

It is obvious that the complexity of this method is pretty high. Next we present algorithm **BBED**(branch and bound for error detection) applying branch and bound method that can greatly reduce the required computations by discarding unsatisfactory branches.

Consider a program of one parameter $F(\mathrm{X}_1)$ and given tolerance ε and δ, we first divide X_1 into $\mathrm{X}_1^{(1)}, \mathrm{X}_1^{(2)}$, where $\omega(\mathrm{X}_1^{(1)}) = \omega(\mathrm{X}_1^{(2)}) = 1/2\omega(\mathrm{X}_1)$, and replace X_1 by $\mathrm{X}_1^{(1)}$ and $\mathrm{X}_1^{(2)}$ respectively, and then run the program to check if error is greater than δ. If the $error \le \delta$, the subdivision will be abandoned. Otherwise, we continue to divide the subdivision until $\omega(\mathrm{X}_1) \le \varepsilon$ andechocurrent subdivision as an unacceptable error-triggering input. Fig. 5 depicts the process of unacceptable error-triggering inputs detection for one program of one parameter.

Algorithm **BBED** is easily generalized to programs of multiple parameters. Suppose given a program $Y = F(\mathrm{X})$, where X and Y are interval vectors and *n* interval-valued inputs $\mathrm{X}_i, i = 1, \cdots, n$, we are aimed at seeking all candidate unacceptable error-triggering inputs $\mathrm{x}_k \subset \mathrm{X}_k$, where x_k is an interval vector, $k \ge 0$. And the algorithm is as follows.

Algorithm BBED. This algorithm echoes all candidate unacceptable error-triggering inputs $x_k \subset X_k, k \geq 0$,stored in □ , where $length(□) = n$.

(1)Initially, the list $L = \{X_1, \cdots, X_n\}$, □ is empty;

(2)Let $X_0 \leftarrow L[1]$, bisect X_0 such that

$X_0 = X_0^{(1)} \cup X_0^{(2)}$;

(3) For $i := 1, 2$ do

(a) $X_0 \leftarrow X_0^{(i)}$;

(b) IF $\boldsymbol{\omega(X_0^{(i)})} > \varepsilon$, then

move X_0 at the tail of L ;

ELSE

remove X_0 from L and place it into □ ;

END IF

(c) execute the program $Y = F(X)$;

(d) IF $error(Y, X) > \delta$, then

IF L is not empty THEN

return to step (2);

ELSE

RETURN with list □ ;

END IF

END IF

END For

Remark. Step (2) means we bisect X_0 into two intervals,

$$X_0^{(1)} = [\underline{X_0}, m(X_0)] \ ,$$

$$X_0^{(2)} = [m(X_0), \overline{X_0}] \ .$$

In step (b), ε represents the condition if the input will be divided into a smaller one. In step (d), $error(Y, X) = \omega(Y) / \omega(X)$,and δ is a tolerance presenting error accumulating ratio.

In the worst case, the complexity of Algorithm **BBED** is $O(2^{n+c})$,where c is some constant depend on the initial of $X_i, i = 1, \cdots, n$ and ε .

EXPERIMENTS

Our experiments are conducted on the machine running Windows 7 with Intel i7-3770 2-Core 3.40GHz, 8GB RAM. And SpaceAED run on Boost 1.50, JDK 1.7.0 and MinGW 2.21. We choose to test functions about matrix arithmetic and special functions of GSL(GNU Scientific Library), whose version is gsl-1.14. We analyze all the functions about matrix arithmetic and most of special functions, even though they highly depend on each other. GSL is a mature, extensively-used, well-tested and well-maintained scientific computation library[14]. It is the reason that we carry on experiments on it. Detecting for unacceptable error-triggering inputs of the functions in GSL is both challenging and important. In spite of challenges, we evaluate SpaceAED over 100 functions in GSL.

We will take three examples to demonstrate how SpaceAED works out in practice.

C. *Example 1*

The program for matrix inversion is made up of 20 functions from GSL. Because there is no direct function used for matrix inversion, we synthesizegsl_matrix_inverse calling gsl_linalg_LU_decomp and gsl_linalg_LU_invert, which are defined in gsl/linalg/lu.c.Fig. 6 depicts implementation of function gsl_matrix_inverse.

```
void gsl_matrix_inverse(gsl_matrix *mat,
gsl_matrix *invm){
 gsl_permutation *p = gsl_permutation_alloc(
                            mat->size1);
int sign = 0;
 gsl_linalg_LU_decomp(mat, p, &sign);
 gsl_linalg_LU_invert(mat, p, invm);
 gsl_permutation_free(p);
}
```

Figure 6. Implementation of function gsl_matrix_inverse.

mat means a pointer to source matrix as the input, and invm means a pointer to inverse matrix as the output.

Here, we initiate mat like this,

$$*mat = \begin{pmatrix} [1,1.2] & [2,2.2] & [1.8,2] \\ [3,3.2] & [4.3,4.5] & [1,1.2] \\ [4,4.2] & [6.4,6.6] & [2.9,3.1] \end{pmatrix},$$

and set $\delta = 10^6$ and $\varepsilon = 0.05$ .Run on this program, SpaceAED takes about 332.07s and returns 4375283 unacceptable error-triggering inputs. In order to verify the correctness of results, we randomly select 10 inputs from the results above and take each of them as the input to compute its condition number, which is used as an index of relative error of the program. Table I shows condition number of the matrix with 10 unacceptable error-triggering inputs.

TABLE I. CONDITION NUMBER OF THE MATRIX WITH 10 UN-ACCEPTABLE ERROR-TRIGGERING INPUTS

Ex.	**Unacceptable Error-triggering Input**	**Condition Number**
1	$\begin{bmatrix} 1.02 & 2.01 & 1.82 \\ 3.03 & 4.31 & 1.023 \\ 4.04 & 6.41 & 2.92 \end{bmatrix}$	492.1753
2	$\begin{bmatrix} 1.02 & 2.02 & 1.81 \\ 3.04 & 4.33 & 1.01 \\ 4.04 & 6.47 & 2.98 \end{bmatrix}$	460.1818
3	$\begin{bmatrix} 1.085 & 2.15 & 1.835 \\ 3.04 & 4.33 & 1.01 \\ 4.04 & 6.47 & 2.98 \end{bmatrix}$	432.9247
4	$\begin{bmatrix} 1.0715 & 2.05 & 1.889 \\ 3.114 & 4.4 & 1.189 \\ 4.16 & 6.48 & 3.1 \end{bmatrix}$	643.9857
5	$\begin{bmatrix} 1.06 & 2.115 & 1.82 \\ 3.03 & 4.38 & 1.03 \\ 4.04 & 6.53 & 3.03 \end{bmatrix}$	1404.5
6	$\begin{bmatrix} 1.14 & 2.14 & 1.935 \\ 3.07 & 4.375 & 1.08 \\ 4.14 & 6.46 & 3.06 \end{bmatrix}$	1683.6
7	$\begin{bmatrix} 1.115 & 2.06 & 1.912 \\ 3.032 & 4.45 & 1.057 \\ 4.12 & 6.49 & 2.949 \end{bmatrix}$	1525.6
8	$\begin{bmatrix} 1.175 & 2.12 & 1.912 \\ 3.014 & 4.45 & 1.017 \\ 4.119 & 6.58 & 3.05 \end{bmatrix}$	503.1505
9	$\begin{bmatrix} 1.04 & 2.13 & 1.96 \\ 3.11 & 4.47 & 1.05 \\ 4.18 & 6.57 & 2.923 \end{bmatrix}$	1050.3

10		$\begin{bmatrix} 1.06 & 2.18 & 1.98 \\ 3.17 & 4.39 & 1.08 \\ 4.19 & 6.46 & 2.935 \end{bmatrix}$	6264.8

D. *Example 2*

Consider the program for computing the eigenvalues and eigenvectors of matrix, void **gsl_eigen_symmv**(gsl_matrix* mat, gsl_vector* eval, gsl_matrix* evec,gsl_eigen_symmv workspace* w) is defined in gsl/eigen/symmv.c. Its implementation depends on 30 other functions in GSL. Here mat means a pointer to source matrix as the input, eval respresents the eigenvalues of mat and evec denotes the eigenvectors of mat as the output.

Here, we initiate mat like this,

$$*mat = \begin{pmatrix} [1,1.2] & [0.1,0.2] & [1,1.2] \\ [0.1,0.2] & [4.3,4.5] & [1,1.2] \\ [1,1.2] & [1,1.2] & [2.9,3.1] \end{pmatrix},$$

and set $\delta = 10^5$ and $\varepsilon = 0.05$. Run on this program, SpaceAED takes 213.82s and reports 1255821 unacceptable error-triggering inputs.

In order to verify the correctness of results, we randomly pick a point *a* from the results above and another point $\tilde{a}$ near *a*, then compute $\|f(\tilde{a}) - f(a)\|_2 / \|\tilde{a} - a\|_2$ as the relative error of this program. TableII shows relative error of gsl_eigen_symmv on point *a*.

TABLE II. RELATIVE ERROR OF GSL_EIGEN_SYMMV ON POINT a

Point a	**Point** $\tilde{a}$	**Relative Error**
$\begin{bmatrix} 1.026 & 0.126 & 1.045 \\ 0.1512 & 4.358 & 1.012 \\ 1.01 & 1.103 & 3.04 \end{bmatrix}$	$\begin{bmatrix} 1.0261 & 0.1265 & 1.0459 \\ 0.15122 & 4.3583 & 1.012 \\ 1.011 & 1.103 & 3.044 \end{bmatrix}$	139.52
$\begin{bmatrix} 1.076 & 0.126 & 1.01 \\ 0.114 & 4.451 & 1.032 \\ 1.11 & 1.180 & 3.084 \end{bmatrix}$	$\begin{bmatrix} 1.07601 & 0.12601 & 1.0101 \\ 0.11401 & 4.45102 & 1.03201 \\ 1.11002 & 1.1801 & 3.08401 \end{bmatrix}$	120.59
$\begin{bmatrix} 1.175 & 0.1 & 1.175 \\ 0.15 & 4.35 & 1.075 \\ 1.05 & 1.0 & 3.01 \end{bmatrix}$	$\begin{bmatrix} 1.17501 & 0.1004 & 1.1751 \\ 0.1504 & 4.3501 & 1.0753 \\ 1.0508 & 1.00001 & 3.01005 \end{bmatrix}$	86.346
$\begin{bmatrix} 1.183 & 0.139 & 1.163 \\ 0.163 & 4.33 & 1.18 \\ 1.029 & 1.084 & 2.91 \end{bmatrix}$	$\begin{bmatrix} 1.1831 & 0.1391 & 1.1631 \\ 0.1631 & 4.3301 & 1.1801 \\ 1.0291 & 1.0841 & 2.9101 \end{bmatrix}$	99.074
$\begin{bmatrix} 1.11 & 0.140 & 1.17 \\ 0.12 & 4.43 & 1.18 \\ 1.13 & 1.095 & 2.913 \end{bmatrix}$	$\begin{bmatrix} 1.1101 & 0.1401 & 1.1701 \\ 0.1201 & 4.4301 & 1.1801 \\ 1.1301 & 1.09501 & 2.9131 \end{bmatrix}$	114.197

TABLE III. RELATIVE ERROR OF GSL_SF_POW_INT ON POINT a

Ex.	**Point** a	**Point** $\tilde{a}$	**Relative Error**
1	2.9375	2.93751	2.13739e+048
2	2.97559	2.97562	7.65513e+048
3	3.14555	3.14557	1.87111e+051
4	3.21289	3.21290	1.5232e+052
5	3.22568	3.22171	1.99691e+052
6	3.74902	3.74912	6.58312e+058
7	4.00391	4.00395	4.42753e+061
8	4.30273	4.30293	5.51652e+064

E. *Example 3*

Consider a function for calculating integer powers in GSL, double **gsl_sf_pow_int** (double x, int n), it is defined in gls/specfunc/pow_int.c and used to compute the power x^n for an integer *n*.

Here we initiate *x*=[2,5] and *n*=20, and then set $\delta = 10^{10}$ and $\varepsilon = 0.005$ .Run on this function, SpaceAED takes 0.026s and echoes 704 unacceptable error-triggering inputs.

In order to verify that each of those inputs do trigger an unacceptable error, we randomly pick a point *a* from the results above and another point $\tilde{a}$ near *a*, and compute as $\left(f(\tilde{a}) - f(a)\right) / \left(\tilde{a} - a\right)$ the relative error of this function,where *f* represents function gsl_sf_pow_int. Table III shows the relative error of gsl_sf_pow_int on point *a*.

CONCLUSION AND FUTURE WORK

In this paper, we have presented the design and implementation of SpaceAED, a tool for automatically detecting all of inputs that trigger unacceptable error. We have also given our extensive evaluation of SpaceAED over 100 GSL functions about Matrix, Vector, Special Function. Experiment results show that SpaceAED is practical, primarily enabled by our combination of program rewriting and technique for efficiently error detection via branch and bound method. Our future work is to releaseSpace-AED open to the public and benefit numerical software developers and users.

Acknowledgment

This work was supported by NSFC(grant 91118007) and Innovation Program of Shanghai Municipal Education Commission (grant 14ZZ046).

References

[1] Wikipedia. Ariane 5 flight 501. http://en.wikipedia.org/wiki/Ariane_5 _Flight_501.

[2] CNN. Toyota: Software to blame for Prius brake problems.http://www.cnn.com/2010/WORLD/asiapcf/02/04/japan.prius.complacomp/index.html.

[3] P.H.Sterbenz, Floating-point Computation.Englewood Cliff, New Jersey: Prentice-Hall,1974, pp. 1-50.

[4] R. E. Moore, R.B. Kearfottand M.J. Cloud, Introduction to Interval Analysis. SIAM, 2009, pp. 5-50.

[5] B. W. Kernighan andD. M. Ritchie, The C Programming Language , 2nd ed..Englewood Cliff, New Jersey: Prentice Hall, 1996, pp. 100-120.

[6] J. M. Muller, N. Brisebarre, F. D. Dinechin, et al. Handbook of Floating-Point Arithmetic.Boston: Birkhäuser Basel, 2010.

[7] J. Clausen, “Branch and Bound Algorithms—Principles and Examples,”Parallel Computing in Optimization, March 1999, pp. 239-267.

[8] J. Levine, flex & bison. Sebastopol, CA: O'Reilly Media, 2009.

[9] Boost C++ Libraries. Boost Interval Arithmetic Library. http://www.boost.org/doc/libs/1_55_0/libs/numeric/interval/doc/interval.html.

[10] P. M. Narendra and K. Fukunaga, “A branch and bound algorithm for feature subset selection,”IEEE Transactions on Computers, Vol. C-26, Sept. 1977, pp. 917-922.

[11] P. M. Narendra and K. Fukunaga, “A branch and bound algorithm for computing k-nearest neighbors,”IEEE Transactions on Computers, Vol. 24, July 1975, pp. 750-753.

[12] R. E.Moore,Interval Analysis.Englewood Cliff, New Jersey: Prentice-Hall, 1966, pp. 50-120.

[13] E. Goubault and S. Putot,“Static analysis of numeric algorithms,” Proc. 13thInternational Static Analysis Symposium (SAS 2006), Springer,Aug. 2006, pp. 18-34.
[14] FSF. GSL: GNU scientific library. http://www.gnu.org/s/g-sl/.
[15] M.Martel,“Propagation of roundoff errors in finite precision computations: a semantics approach,”Proc.11th European Symposi-
um on Programming (ESOP2002), Springer, April 2002, pp. 194-208.

APPENDIX

By an *interval*, we adopt*X*denoting it, and the left and right endpoints of an interval*X*will be denoted by $\underline{X}$ and $\overline{X}$ respectively. Thus,

$$X = [\underline{X}, \overline{X}].$$

The basic arithmetic operations between intervals are, for $[\underline{X}, \overline{X}]$and $[\underline{Y}, \overline{Y}]$,

$$[\underline{X}, \overline{X}] + [\underline{Y}, \overline{Y}] = [\underline{X} + \underline{Y}, \overline{X} + \overline{Y}],$$
$$[\underline{X}, \overline{X}] - [\underline{Y}, \overline{Y}] = [\underline{X} - \overline{Y}, \overline{X} - \underline{Y}],$$
$$[\underline{X}, \overline{X}] \cdot [\underline{Y}, \overline{Y}] = [\min(\underline{X}\,\underline{Y}, \underline{X}\,\overline{Y}, \overline{X}\,\underline{Y}, \overline{X}\,\overline{Y}),$$
$$\max(\underline{X}\,\underline{Y}, \underline{X}\,\overline{Y}, \overline{X}\,\underline{Y}, \overline{X}\,\overline{Y})],$$
$$[\underline{X}, \overline{X}] / [\underline{Y}, \overline{Y}] = [\min(\underline{X} / \underline{Y}, \underline{X} / \overline{Y}, \overline{X} / \underline{Y}, \overline{X} / \overline{Y}),$$
$$\max(\underline{X} / \underline{Y}, \underline{X} / \overline{Y}, \overline{X} / \underline{Y}, \overline{X} / \overline{Y})], \text{when } 0 \notin [\underline{Y}, \overline{Y}].$$

The *width* of an interval*X* is denoted by

$$\omega(X) = \overline{X} - \underline{X} \ .$$

The *midpoint* of an interval *X*is denoted by

$$m(X) = 1/2(\overline{X} + \underline{X}).$$

By an *n-dimensionalinterval vector*, we denote it by an *n*-tuple of intervals

$$(\mathrm{X}_1, \cdots, \mathrm{X}_n),$$

and we will also adopt X denoting interval vectors.

The *width* of an interval vector $\mathrm{X} = (\mathrm{X}_1, \cdots, \mathrm{X}_n)$ is the largest of the widths of all its component intervals,

$$\omega(X) = \max_i \omega(X_i).$$

Interval arithmetic can be extensively used in scenario where no exact numerical values can be stated. It is often used to handle error analysis, namely to keep track of rounding errors arising from each calculation because it uses an interval that contains the true result. And rounding error brought by the cur-rent calculation is given by[12],

$$error := |b - \mathrm{a}| \text{ for } [\mathrm{a}, b].$$