Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2017-IJ-002

2017-IJ-002

Software Numerical Instability Detection andDiagnosis by Combining Stochastic andInfinite-precision Testing

Enyi Tang, Xiangyu Zhang, Norbert Th. Muller, Zhenyu Chen, and Xuandong Li

IEEE Transactions on Software Engineering 2017

Software Numerical Instability Detection and Diagnosis by Combining Stochastic and Infinite-Precision Testing

Enyi Tang, Xiangyu Zhang, Norbert Th. Müller, Zhenyu Chen, and Xuandong Li

Abstract—Numerical instability is a well-known problem that may cause serious runtime failures. This paper discusses the reason of instability in software development process, and presents a toolchain that not only detects the potential instability in software, but also diagnoses the reason for such instability. We classify the reason of instability into two categories. When it is introduced by software requirements, we call the instability *caused by problem*. In this case, it cannot be avoided by improving software development, but requires inspecting the requirements, especially the underlying mathematical properties. Otherwise, we call the instability *caused by practice*. We design our toolchain as four loosely-coupled tools, which combine stochastic arithmetic with infinite-precision testing. Each tool in our toolchain can be configured with different strategies according to the properties of the analyzed software. We evaluate our toolchain on subjects from literature. The results show that it effectively detects and separates the instabilities caused by problems from others. We also conduct an evaluation on the latest version of GNU Scientific Library, and the toolchain finds a few real bugs in the well-maintained and widely deployed numerical library. With the help of our toolchain, we report the details and fixing advices to the GSL buglist.

Index Terms—Numerical analysis, infinite-precision arithmetic, stochastic arithmetic, software testing

1 Introduction

The rapid advancement of technology makes modern personal computers and devices powerful in computing. Hence, complex numerical algorithms become affordable and are widely applied in our daily software. Ensuring the correctness of numerical computing is important for software reliability. On the other hand, it is also very challenging. In particular, the unique and complex computation logic in numerical computing often poses new challenges in software testing and verification. Furthermore, truncations due to the limited representation precision happen all the time in numerical computing so that numerical errors are inevitably accumulated during propagation of values. Thus, for modern numerical software, we need not only methods of numerical analysis, but also tools that help us build and test numerical programs, especially their stabilities in the presence of numerical errors.

Given its importance, a lot of recent work tries to address the problem from different aspects. For example, Jézéquel et al. build a tool called CADNA [1], which automatically detects numerical instabilities with a stochastic method. Martel implements an optimizer that statically transforms numerical programs to more stable forms [2], [3]. Barr et al. [4] analyze the relationship of inputs and numerical exceptions by symbolic execution. Based on these researches, we present a new toolchain of numerical testing in this paper, which is designed according to the modularity principles in software development process. Different from the existing tools, our toolchain not only automatically detects the potential instabilities in numerical programs, but also diagnoses the reason for the instabilities.

Software development process is often composed of a few distinct phases such as requirement analysis, software design, implementation, software testing, deployment and maintenance. Numerical instabilities can be introduced at any phases before software testing. Here we define numerical instabilities as *problems in which internal errors caused by truncation or external input errors due to precision limitation in raw data collection lead to substantial output variations*. In the presence of instabilities, it is often hard to trust the data processing results. Our toolchain focuses on testing instabilities, Fig. 1 presents the various aspects in numerical computing that cause instabilities. The mathematical properties of the problem aimed by the software may entail instabilities. In this case, the instabilities cannot be avoided by improving software development, so we call them numerical instabilities *caused by problem*. Improper software design and implementation can also introduce instabilities. We call them numerical instabilities *caused by practice* in our toolchain, because we can fix them by improving our development

- E. Tang, Z. Chen and X. Li are with the State Key Laboratory for Novel Software Technology and Software Institute of Nanjing University, Jiangsu 210023, China. E-mail: {eytang, zychen, lxd}@nju.edu.cn.
- X. Zhang is with the Department of Computer Science, Purdue University, 305 North University Street, West Lafayette, IN 47907-2107. E-mail: xyzhang@cs.purdue.edu.
- N. Th. Müller is with Abteilung Informatik, University of Trier, D-54286 Trier, Germany. E-mail: mueller@uni-trier.de.

Manuscript received 29 Oct. 2015; revised 19 Sept. 2016; accepted 19 Nov. 2016. Date of publication 20 Dec. 2016; date of current version 23 Oct. 2017.
Recommended for acceptance by S.F. Siegel.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2642956

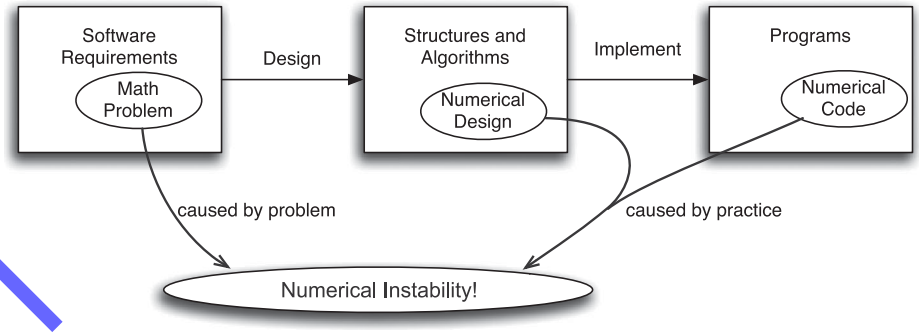


Fig. 1. Categories of aspects that cause numerical instabilities in software.

practice. As well as detecting numerical instabilities in software, our toolchain automatically diagnoses them by categorizing the instabilities. The diagnostic information is useful for developers to fix the instabilities. If the instabilities are caused by problem, developers should think about improving their software at a higher level, which means redefining the software requirements to mitigate or get around the instability inducing mathematical properties. Otherwise they can improve the software itself with better numerical design and programming practices.

The insight of our toolchain is that we combine the techniques of stochastic and infinite-precision testing. Stochastic testing estimates the accumulated numerical errors during the propagation of values in software. While we randomly introduce small changes in numerical computation, the fluctuation range of program outputs can help us estimate the magnitude of numerical errors accumulated during the execution. Then we use infinite-precision arithmetic (IPA) to generate the exact output of numerical computing to show the properties of mathematical problems and diagnose the source of instabilities—caused by problem or by practice. The infinite-precision arithmetic iterates the computing process to make sure each numerical output is precise enough for the computing logic. Different from traditional floating-point arithmetic with a fixed number of digits to represent a value, infinite-precision arithmetic works by increasing the number of digits on demand through dynamic memory allocation. It has an iteration module to make sure the allocated memory is larger enough for every numerical value in the program.

For flexibility, we design our toolchain as four loosely-coupled tools: `ipatrans`, `fpstoc`, `ipstoc` and `ediagno`. The tool `ipatrans` first tranforms numerical programs to the infinite-precision arithmetic, which can produce the exact output for every numerical computation and perform as the test oracle in our testing. Then `fpstoc` and `ipstoc` separately apply the stochastic perturbation on both the fixed-precision floating-point arithmetic (FPFA) and the infinite-precision arithmetic. The result of perturbing FPFA shows us if the program implementation is stable, while perturbation on IPA shows if the instability is caused by the mathematical properties of the problem. Finally, `ediagno` statistically synthesizes the data together to provide the conclusions. When the program is unstable and it is caused by practice, the toolchain also provides localization and fixing hints of the instabilities. The flexible design of our toolchain is convenient for applying new strategies in testing. For example, users can apply other strategies of stochastic or infinite-precision arithmetic in the toolchain. We evaluate our toolchain on a set of test subjects from the literature, and also the GNU Scientific Library. The results show that the toolchain is able to detect and diagnose the instabilities in real-world programs, and further find real bugs in the latest version of GNU Scientific Library. With the help of our toolchain, we report the details and fixing advices to the buglist of the library.

The major contributions of this paper are the following.

- We design a numerical testing toolchain that closely couples with the process of software development. The toolchain is able to detect numerical instabilities in software and diagnose the reason in causing such instabilities.
- We implement our toolchain with four loosely-coupled tools by combining the techniques of stochastic and infinite-precision testing. The implementation is flexible and easy to be updated with different strategies.
- We evaluate our approach on a few test subjects from the literature and also the GNU Scientific Library. Our toolchain successfully detects and diagnoses instabilities in the subjects. The newly identified bugs are reported to the buglist of the subjects.

The rest of this paper is organized as follows: Section 2 introduces a number of definitions related to instability and presents an illustrative example in industry that uses our toolchain. Then in Section 3, we discuss the technical details of the toolchain. Evaluation results and their discussion can be found in Section 4. Section 5 shows the related work of this paper. And finally we give our conclusion and talk about some future work in Section 6.

2 Definitions and an Example

This section first introduces a number of definitions related to numerical instability, such as *instability caused by problem* and *instability caused by practice*. We also show insights of the differences between the two kinds of instabilities. Then we provide a real world example to illustrate the usage of our techniques.

2.1 Definitions Related to Instability

Software engineers develop numerical software to solve numerical problems. Formally, we denote the problem with a mathematical function f that maps the input x to its solution $f(x)$. Software engineers also need an algorithm to solve the problem f. In this paper, we denote the numerical

algorithm over the input x with a finite sequence of steps $f_1, f_2, f_3, \ldots, f_n$, where $f(x) = f_n(f_{n-1}(\ldots f_2(f_1(x))\ldots)$.[1] Let $\hat{f}$ be a numerical program that implements f with the corresponding algorithm. Hence we have $\hat{f}(x) = \hat{f}_n(\hat{f}_{n-1}(\ldots \hat{f}_2(\hat{f}_1(x))\ldots)$, where $\hat{f}_i$ is the numerical code that implements the step $f_i (1 \leq i \leq n)$ in the algorithm.

When small changes at the input or at an intermediate value, which means the value provided to a step $\hat{f}_i$, cause $\hat{f}(x)$ to change substantially, we say the program $\hat{f}$ is *unstable*, or the program contains *instability* problems. Such instability might be a property of the mathematical function itself. For example, the mathematical solution $f(x)$ changes substantially when the argument x changes just a little. In this case, no matter how we improve the algorithm or the implementation to make the program $\hat{f}$ more precise to the function f, the instability must always exist since it is an essential property of f. Instabilities in a numerical program can also be caused by the implementation. For example, when truncation errors accumulate drastically in a numerical algorithm, the program becomes unstable. But sometimes experts use a stable algorithm to avoid such instabilities because such instabilities depend on the algorithm or the implementation. Generally, if the program $\hat{f}$ is unstable when small changes at the argument x or the corresponding algorithm step f_i also cause the mathematical solution $f(x)$ to change a lot, we say the instability is *caused by problem*. Otherwise, we say the instability is *caused by practice*.

A typical example of instability caused by problem is the ill-conditioned problem, where small input changes cause the solution to change a lot. No matter which algorithm or implementation software engineers choose, the instability must always exist in the program that implements such a problem. In this case, software engineers can only persuade users to compute other problems that satisfy their requirements or devote more computing resources in computing such a problem and make sure the results are correct.

Instabilities caused by practice also occur in the implementations of numerical programs. For example, because the truncation errors accumulated in some program steps (such as $\hat{f}_i$) grow quickly, the program $\hat{f}$ becomes unstable. When we examine only the underlying mathematical problem $f(x)$ without taking the truncation errors into account, the instability cannot be detected. Such instability problems can be fixed by a new implementation in computing $\hat{f}$ that goes through the unstable step $\hat{f}_i$. Specifically, software engineers should substitute the subsequence of steps around $\hat{f}_i$ (together with $\hat{f}_i$ itself) with another mathematical equivalent but stable implementation $f'_1, f'_2, \ldots, f'_m$.

Our toolchain detects instabilities in numerical software and diagnoses if it is caused by practice with two measurements, the *implementational condition number* (ICN) and the *statistical condition number* (SCN). Intuitively, ICN measures how the program output $\hat{f}(x)$ changes along with small perturbations of inputs or intermediate values, whereas SCN measures how the problem solution $f(x)$ changes. Hence, a large ICN means the program is unstable. Furthermore, a large SCN implies the instability is caused by problem, whereas a large ICN with a small SCN suggests the instability is caused by practice.

1. In practice, the algorithm of f may consist of complicated control flow such as branches and loops. However for a specific input x, we always get a deterministic trace of the algorithm. Here we consider a trace as a finite sequence of steps and treat an algorithm as a set of traces corresponding to different inputs.

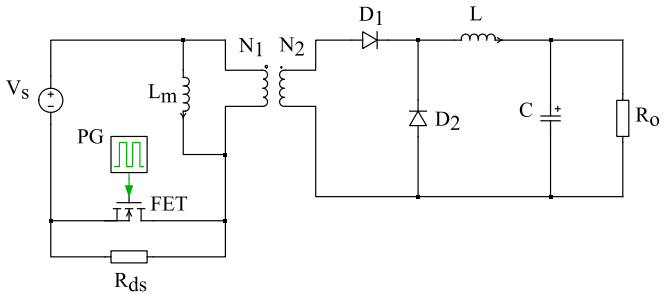


Fig. 2. Circuit of a forward converter with error.

2.2 An Illustrative Example

Different types of numerical instabilities exist in the real world, hence our technique ought to be practical and useful. Here we provide a real world example from the area of power electronics, and introduces the usage of our toolchain. The requirement of the example is derived from circuit design. Designers often use numerical programs to simulate their circuit before real manufacture. Problems in the circuit itself as well as improper implementation of the simulation program may cause numerical instabilities. Our toolchain detects instabilities in simulation, and further diagnoses if it is a requirement problem (from the circuit specification), or a coding problem (from the software implementation).

Fig. 2 presents the circuit of a forward converter designed by some amateur in power electronics. Forward converter is a kind of hardware that uses a transformer to increase or decrease the DC (Direct Current) voltage [5]. The circuit in Fig. 2 has an error. It is a common practice to add a magnetizing inductance L_m across an ideal transformer to simulate the effects of real winding. However in this case, it is wrong to have the inductance, causing numerical instabilities during simulation.

Since the physical characteristics of the circuit is beyond the scope of our paper, we provide the key simulation code that induces the numerical instability in Fig. 4a. The code snippet computes the voltage across the magnetizing inductance L_m of the circuit. The simulation is set up by providing the specific parameter values that correspond to the elements in the circuit. Unfortunately, the simulation is unstable even with the default setup.

By default, the circuit simulation is set up with the voltage source $V_s = 325$ V, the magnetizing inductance $L_m = 5.0\ \mu$H, the ideal transformer with the windings $N_1 : N_2 = 12 : 1$, the filter inductance $L = 50.0\ \mu$H, the capacitance $C = 400.0\ \mu$F, and the output resistance $R_o = 1.0\ \Omega$. For the metal-oxide-semiconductor field-effect transistor (MOS-FET) that serves as a switch in the circuit, the circuit designer uses a drain-source resistance $R_{ds} = 8.0\ \text{M}\Omega$ to simulate the leakage of the cut-off state in the MOS-FET. The control signal for the MOS-FET is generated by the pulse generator (PG) as a square wave input with the frequency of 20 kHz, and the duty cycle (duty ratio) $D = 0.4$. Hence in the simulation code, the variable values are `Vs = 325, Lm = 5E-6, D = 0.4, Period = 5E-5, Acd = 1.0, Gds = 1.25E-7 +Acd, N1 = 12, N3 = 12`.

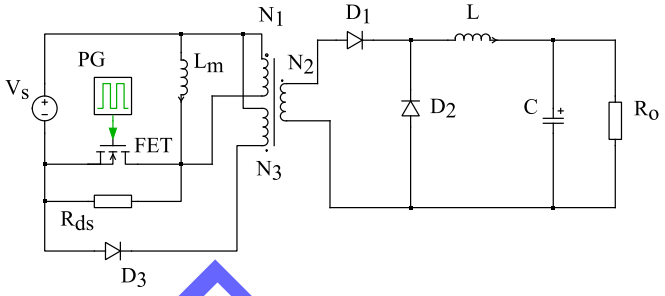


Fig. 3. Circuit of the corrected forward converter.

When analyzing the code in Fig. 4a with our toolchain, we get ICN = 6.751×10^6 and SCN = 6.215×10^5. From the result, the code in Fig. 4a is unstable and the instability is from the software requirement. Particularly, when the code simulates the behavior of the circuit, it needs to compute the voltage in a continuous time period. So the input `time` can be any value in its domain, including a value near `D*Period`. This causes serious massive cancellation and the error is further enlarged by the later calculation, which makes the program particularly unstable. As shown in Fig. 5a, the massive cancellation is caused by an impulse voltage in the circuit. Hence, no matter how accurate the code is, the instabililty always exists in the simulation. As software engineers cannot fix the problem, they ask the circuit designer to inspect the requirements (the circuit).

The circuit designer corrects the error with a new circuit [6], which is shown in Fig. 3. The simulation code that computes the voltage across the magnetizing inductance L_m of the new circuit is in Fig. 4b. However, the code is still unstable due to improper implementation, since the last operation of the expression at line 6 may also cause cancellation when `time` is close to `(N1+N3)*D*Period/N1`, together with the expression `D*Period/(Gds-Acd)-Lm` that subtracts a small number from a large one. When analyzing the new code in Fig. 4b with our toolchain, we get ICN = 1.478×10^7 and SCN = 0.00917, which indicates that

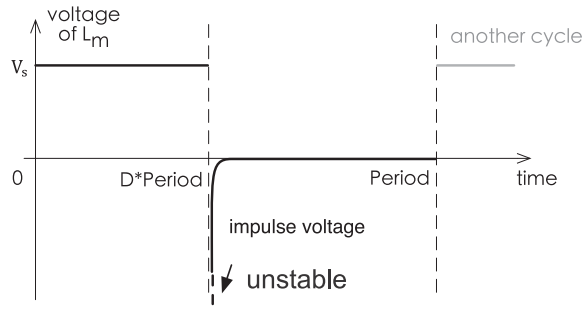
```
1  float VoltageLm_CircuitErr(float time,
2      float Vs, float Lm, float D,
3      float Period, float Gds, float Acd)
4  {
5      float v1=(D*Period-time)/(Gds-Acd)/Lm;
6      float v2=Vs/Lm*(Lm-D*Period/(Gds-Acd))
7                    *exp(v1);
8      return v2;
9  }
```

(a) Error Circuit Voltage

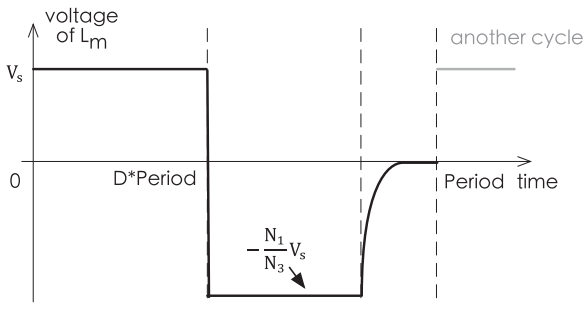
```
1  float VoltageLm_CircuitRight(float time,
2      float Vs, float Lm, float D,
3      float Period, float Gds, float Acd,
4      int N1, int N3)
5  {
6      float v1=(N1+N3)*
7          (D*Period/(Gds-Acd)-Lm)/(N1*Lm)
8          -1.0/(Gds-Acd)/Lm*time;
9      float v2=(N1*exp(v1)+N3)/N3;
10     return Vs*(1-v2);
11 }
```

(b) Corrected Circuit Voltage

Fig. 4. Simplified simulation code of computing the voltage across the magnetizing inductance L_m in the circuits.



(a) Error Circuit



(b) Corrected Circuit

Fig. 5. Theoretical voltage across the magnetizing inductance L_m in a cycle of the circuits.

the instability is caused by practice and can be mitigated or avoided by better implementation. Such result is in accordance with the theoretical values in Fig. 5b, which shows that the impulse voltage problem has been avoided. Our toolchain also provides hints for fixing the numerical implementation. With the hints, the software developer replaces the numerical expression at lines 6-8 with `(Period*D*N1/(Gds-Acd)/N1+Period*D*N3/(Gds-Acd)/N1-time/(Gds-Acd))/Lm-Lm*N1/Lm/N1-Lm*N3/Lm/N1`, which is mathematically equivalent to the expression in Fig. 4b but makes the code numerically stable. The ICN of the replaced expression is 2.119.

In short, the instability in Fig. 4a is caused by problem because the root cause is the impulse voltage shown in Fig. 5a. As the impulse voltage is a physical characteristic of the circuit, we cannot improve it by just changing the implementation. In contrast, the instability in Fig. 4b is caused by practice because the improper code causes it. Only when both kinds of instabilities are avoided, the numerical computation becomes stable.

3 Approach and Implementation

This section presents the technical details of our toolchain.[2]

3.1 Main Workflow

Fig. 6 presents the main workflow of our toolchain. It contains four loosely-coupled tools that work in sequence: `ipatrans`, `fpstoc`, `ipstoc` and `ediagno`. The source code of the subject program that mainly contains fixed-precision floating point arithmetic is first provided to `ipatrans`. The tool transfers the fixed-precision floating point

2. The toolchain is available at http://seg.nju.edu.cn/~eytang/numericaltoolchain.tar.gz

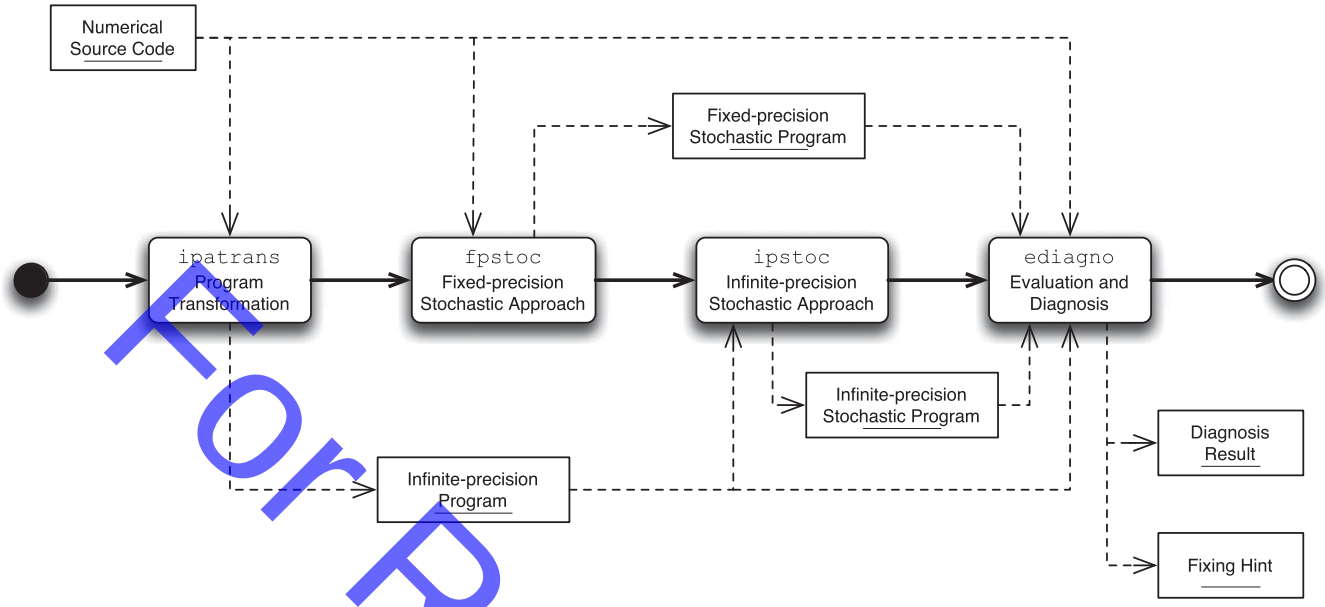


Fig. 6. Main workflow of the numerical testing toolchain.

arithmetic in the program to infinite-precision arithmetic automatically. Although the transferred infinite-precision program is much slower than the original one, it is precise and acts well as our test oracle in the later step for instability diagnosing. Then we provide two stochastic transformation tools: `fpstoc` and `ipstoc`. The tool `fpstoc` extends the fixed-precision floating point arithmetic with a stochastic approach, while `ipstoc` performs similar transformation on the infinite-precision program. A few different strategies such as the CESTAC method [7], [8] can be applied in this step. For generalization we choose the plain strategy in our implementation. The plain strategy systematically alters each input and intermediate value with uniformly distributed random numbers in a small but controllable range around it. Although the CESTAC method is more efficient, the plain strategy directly changes the values and provides the exact statistical results in general situation. With the transformation of `fpstoc` and `ipstoc`, the transformed programs output the results with random perturbation on numerical values during execution. We hence call them stochastic programs. The infinite precision stochastic program introduces random values at each algorithm step, but computes every step precisely without rounding errors. Hence it shows the underlying mathematical properties of the problem. However, the fixed-precision stochastic program introduces random values with rounding errors at each step, which shows the properties of the program. The tool `ediagno` drives the testing procedure and generates the statistical results based on the outputs of the execution of the transformed programs. If the outputs of the fixed-precision stochastic vary a lot from the original results, the numerical software is unstable. In this case, `ediagno` checks the corresponding outputs of the infinite-precision stochastic program. Since infinite-precision arithmetic produces the exact results of the algorithm, the outputs of the infinite-precision stochastic program explain the property of the mathematical problems implied in the software requirements. Particularly, if the outputs of the infinite-precision stochastic program also vary a lot, the numerical instabilities in the software are caused by the ill-conditioned problems implied in the software requirements. In this case, developers should ask the software customers to inspect the requirements. Otherwise, the numerical instabilities in the software are caused by practice, and `ediagno` will also provide some hints for fixing the numerical implementation of the software.

3.2 `ipatrans`: Infinite-Precision Transformation

The goal of using infinite-precision arithmetic is to build a system that can give us the *precise* numerical outputs without truncation errors. In other words, every significant digit of the output must be correct in this arithmetic. To fulfill the goal, we use the arbitrary precision floating-point format [9] (`type_real`) to represent values in the program. Fig. 7 denotes the basic format of the arbitrary precision floating-point numbers. It consists of a sign bit, a few exponent bits and significand bits. When s is the sign; b is the base with the value 2; l is the number of digits in the significand; x is the exponent; and z is the integer value of the numerator of the significand, we have the value

$$(-1)^s \times \frac{z}{b^{l-1}} \times b^x. \quad (1)$$

Different from the fixed precision floating-point format, the arbitrary precision format dynamically allocates memory of significand bits during execution. As such, users can extend the number of bits for the value when needed. The precision of such floating-point numbers is not bounded whenever

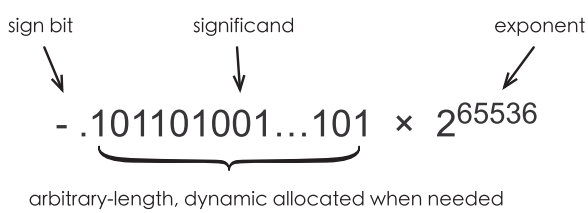


Fig. 7. Format of arbitrary precision floating-point numbers used in infinite-precision arithmetic.

there is still available memory space. However, it lacks a mechanism to indicate how much memory is sufficient for a value, or what is the proper condition of increasing precision.

The infinite precision arithmetic extends the arbitrary precision format with an iteration, to make sure each arbitrary precision floating point value has sufficient bits to represent the precise value. The iteration in the infinite precision arithmetic refines the values with increasing significand length, so the intermediate and output values are closer to the precise results after each iteration. The iteration does not terminate until each represented value $\hat{v}$ is close enough to the corresponding precise value v. Formally, when the infinite precision arithmetic iterates in the computing process, it outputs a sequence $\hat{v}_1, \hat{v}_2, \hat{v}_3, \ldots, \hat{v}_n$. Each output $\hat{v}_i$ is the result using higher precision of computing than the previous one $\hat{v}_{i-1}$. The sequence gradually approximates the precise result v. According to the definition of the precise result of the program, we have

$$v = \lim_{n \to \infty} \hat{v}_n. \tag{2}$$

According to the definition of sequence limit, we can infer that the iteration in infinite precision arithmetic is convergent.

Theorem 3.1 (Convergence of Infinite-precision Iteration). *For every $\varepsilon > 0$, there is a corresponding integer N such that*

$$\text{if} \quad n_1 > N \wedge n_2 > N \quad \text{then} \quad |\hat{v}_{n1} - \hat{v}_{n2}| < \varepsilon. \tag{3}$$

Proof. According to Equation (2) and the definition of sequence limit, we can draw the conclusion that for every $\varepsilon' > 0$ there is a corresponding integer N such that

$$\text{if} \quad n > N \quad \text{then} \quad |\hat{v}_n - v| < \varepsilon',$$

Let $\varepsilon = 2\varepsilon'$, we derive

$$\begin{aligned} \text{if} \quad n_1 > N \wedge n_2 > N \quad \text{then} \\ |\hat{v}_{n1} - \hat{v}_{n2}| &= |(\hat{v}_{n1} - v) - (\hat{v}_{n2} - v)| \\ &\leq |\hat{v}_{n1} - v| + |\hat{v}_{n2} - v| \\ &< \varepsilon' + \varepsilon' = \varepsilon. \end{aligned}$$

□

Since we may not get the precise result v during the iteration, Theorem 3.1 shows that we can terminate when the computed outputs get close enough.

Algorithms 1 and 2 show our implementation of the iteration in infinite precision arithmetic [10]. The basic idea of the iteration is that since program outputs are eventually serialized to output devices, such as the screen, disk and network, our algorithms monitor the serialization functions to obtain the output $\hat{v}$ for each iteration. Moreover, the serialization functions also provide the value ε, which determines the stop conditions of the iteration. Since users often emit the first n digits in their results of real numbers, the length of serialization provides us information of output precision that user defined. In Algorithm 1, ε is defined with the formula $b^{-n} \times |\hat{v}|$, where b is the base of the serialized output. It is often set to 10 when the output is decimal. When the program emits n digits of the result, any error less than ε in the result cannot be shown in the serialized string. Algorithm 1 also holds a static queue V to store a sequence of $\hat{v}_n$, and make sure the sequence is convergent. By comparing the differences of value $\hat{v}$ and other values in the queue V, the algorithm triggers iteration by the exception mechanism. Algorithm 2 presents the iterative process in the infinite precision arithmetic. The computing processes of the numerical program are summarized in the function `program_exec()`. When it catches the exception thrown by the serialization functions, it allocates more memory for the values and increases the length of the significand. After that, it repeats the process until all outputs are sufficiently precise (convergent).

Algorithm 1. `serial`: The Serialization Function of Infinite Precision Arithmetic

input: $n \in$ **Integer** // serialization length
$\hat{v} \in \mathscr{R}$ // serialization value of current iteration, a computable real number
output: $sv \in$ **String** // serialized string output
1: **static Queue** V
2: $\varepsilon \leftarrow b^{-n} \times |\hat{v}|$
3: **if** !`is_full`(V) **then** // store the sequence of $\hat{v}$
4: V.enqueue$(\hat{v})$
5: **throw** Exception re_iterate
6: **end if**
7: **if** $\exists \hat{v}' \in V, |\hat{v} - \hat{v}'| \geq \varepsilon$ **then** // not convergent
8: **do**{}**until** V.dequeue() == $\hat{v}'$
9: V.enqueue$(\hat{v})$
10: **throw** Exception re_iterate
11: **else** // convergent
12: $sv \leftarrow$ `IO`$(\hat{v}, n)$
13: **end if**
14: **return** sv

Algorithm 2. `iteration_wrap`: The Iteration Process of Infinite Precision Arithmetic

1: **repeat**
2: **try**
3: `program_exec()` // the original computing
4: rerun ← `false`
5: **catch** Exception re_iterate
6: `increase_precision()`
7: rerun ← `true`
8: **end try**
9: **until** rerun == `false`

The tool `ipatrans` transforms a fixed-precision floating point program to the version with infinite precision. It is implemented on `Clang`, an open source C/C++ compiler. It works by traversing and transforming the abstract syntax trees (AST) of the program. Algorithm 3 describes the transformation process, which mainly consists of four parts: type substitution, operation modification, serialization and iteration wrapper. It first substitutes all fixed-precision floating point types such as `float` and `double` to the arbitrary precision type `type_real` at all places of definition. Runtime support is also added to dynamically increase the digits used to represent `type_real` values on demand. The

```
1  double p_fix (double value, int bits){
2    byte *byte_array = (byte*)(&value);
3    for(i = 0; i < bits/8; ++i){
4      byte_array[i] ^= (byte)rand();
5    }
6    byte_array[i] ^= (0xFF >> (8-bits%8)) & ((byte)rand());
7    return value;
8  }
```

Fig. 8. Plain stochastic function for fixed-precision arithmetic.

algorithm also replaces the operations on the substituted variables to ensure semantic consistency, including unary operations such as `fabs`, `sin`, and `exp`, and binary operations such as $\div$ and `pow`. Algorithm 3 also adds the `serial` function defined in Algorithm 1 at each input and output point to check the convergence of iterations with the acceptable error ε. Then it wraps the original numerical code body to the iteration defined in Algorithm 2. Specifically, it transfers the `main` function of the original numerical code to the `program_exec()` in Algorithm 2 and wraps it with the exception handler to repeat the computation.

Algorithm 3. Transformation Algorithm from Fixed Precision to Infinite Precision Arithmetic

input: $\mathcal{P}_{\text{fix}}$ // the original numerical program
output: $\mathcal{P}_{\text{inf}}$ // the transformed infinite-precision program
1: $\mathcal{P} \leftarrow \mathcal{P}_{\text{fix}}$ [`main` $\mapsto$ `program_exec`]
2: $\forall\, \eta \in \text{AST}(\mathcal{P})$ **do** // traverse the abstract syntax tree
3: **switch** ($node(\eta)$) // substitute types and operations
4: **case** variable_definition
5: **if** $type(\eta) =$ `double` $\vee\, type(\eta) =$ `float` **then**
6: $\mathcal{P} \leftarrow \mathcal{P}[type(\eta) \mapsto$ `type_real`$]$
7: **end if**
8: **case** function_definition
9: **if** $return(\eta) =$ `double` $\vee\, return(\eta) =$ `float` **then**
10: $\mathcal{P} \leftarrow \mathcal{P}[return(\eta) \mapsto$ `type_real`$]$
11: **end if**
12: **if** $para(\eta) =$ `double` $\vee\, para(\eta) =$ `float` **then**
13: $\mathcal{P} \leftarrow \mathcal{P}[para(\eta) \mapsto$ `type_real`$]$
14: **end if**
15: **case** unary operator f : `type_fix` $\rightarrow$ `type_fix:`
16: $\mathcal{P} \leftarrow \mathcal{P}[f \mapsto f_{\text{inf}}]$ // where f_{inf} : `type_real` $\rightarrow$ `type_real`
17: **case** binary operator $\odot$: `type_fix` $\times$ `type_fix` $\rightarrow$ `type_fix:`
18: $\mathcal{P} \leftarrow \mathcal{P}[\odot \mapsto \odot_{\text{inf}}]$ // where $\odot_{\text{inf}}$: `type_real` $\times$ `type_real` $\rightarrow$ `type_real`
19: **case** `IO`$(\hat{v}_{fix}, n)$ // add the serialization function
20: $\mathcal{P} \leftarrow \mathcal{P}[\eta \mapsto$ `serial` $\hat{v}, n)]$
21: **endswitch**
22: **end for**
23: $\mathcal{P}_{\text{inf}} \leftarrow$ `iteration_wrap` $\mathcal{P}$) // wrap the iteration process
24: **return** $\mathcal{P}_{\text{inf}}$

3.3 `fpstoc` & `ipstoc`: Stochastic Transformation on Fixed-Precision and Infinite-Precision Arithmetic

The stochastic approach estimates the accumulated numerical errors during computation. In our toolchain, we apply the stochastic approach on both the original fixed-precision program and its infinite-precision version. This allows us to measure the sensitivity of both the numerical program and the underlying mathematical problem (to small errors in the inputs and mediate values). Because small errors can cause substantial output changes in sensitive programs, such programs are considered unstable. The fixed-precision stochastic approach detects instabilities in the program implementation, whereas the stochastic results of the infinite-precision program indicate if the instability is caused by the mathematical properties of the problem statement in the software requirements.

The user can choose different stochastic strategies in this phase, such as the plain stochastic strategy, or the CESTAC stochastic strategy [7], [8]. The plain stochastic strategy directly perturbs all the inputs and intermediate values in the program with a predetermined scale, whereas the CESTAC strategy works in a more efficient way, with the hypothesis that the individual round-off errors of the floating-point arithmetic are independently random and uniformly distributed. With the approximation of a probabilistic first order model in 2^{-l} (Equation (1)), the CESTAC approach only needs about three different executions to conclude the stability of the implementation.

In general, we formalize the stochastic approach like this: For each value v in the program, we simulate the error with a basic stochastic function p, and changes it to the value v'

$$v' = p(v, m) = v + v \times \delta \times m = v(1 + \delta m), \tag{4}$$

where m is the order of magnitude in the stochastic approach. In different stochastic stategies, the order of magnitude is expressed as different forms. For example, the plain stochastic stategy randomly alters the k tail bits in the significand, so the value $m = b^{k-l+1}$, while the CESTAC stategy fixes the order of magnitude with the value of ulp (unit in the last place), where $m = b^{-l}$. In Equation (4), $\delta \in \mathbb{R}$ is a random real number taken from a uniform distribution between -1 and 1. Hence, each execution of the basic stochastic function changes the value v with the precentage of δm.

Figs. 8, 9, 10, and 11 presents the implementation of basic stochastic functions under different strategies. Figs. 8 and 9 describe the plain stochastic strategy on a single floating-point value. They both accept an extra parameter to regulate the number of bits to be perturbed. Fig. 8 works on a double precision floating-point number with the fixed length of 64 bits for each value, while Fig. 9 works on an infinite-precision value, which dynamically allocates the significand by an unsigned array (the array `limb` in Fig. 9). The unsigned array stores significand in the little endian order, which means the least significant unit is placed at the index of 0 of the array. So the stochastic function changes the bits from index 0 to the index of the argument. The CESTAC strategy mainly switches the rounding mode to fulfill the stochastic procedure. Fig. 10 presents the basic stochastic function with the CESTAC strategy on a fixed-precision floating-point value.

```
real p_inf (real infvalue, int bits){
  unsigned *limb = infvalue.significand;
  for(i = 0; i < bits/sizeof(unsigned); ++i){
    limb[i] ^= (unsigned)rand();
  }
  limb[i] ^= (((unsigned)(~0)) >> (sizeof(unsigned)-bits%sizeof(unsigned))) &
                ((unsigned)rand());
  return infvalue;
}
```

Fig. 9. Plain stochastic function for infinite-precision arithmetic.

```
double p_fix (double value){
  // just directly change the rounding
  // mode by low-level floating point
  // instructions every time when
  // reading a value, so every opera-
  // tion will be perturbed in its
  // result with different rounding.

  cestac_random_roundmode();
  return value;
}
```

```
void cestac_random_roundmode (){
  int rn ← rand();
  if(rn <= RAND_MAX/2){
    round_to_plus_infinity();
  }else{
    round_to_minus_infinity();
  }
}
```

Fig. 10. Fixed-precision basic stochastic function with the CESTAC strategy.

```
real p_inf (real infvalue){
  unsigned* limb = infvalue.significand;
  int shift_num = (infvalue.significand_size*sizeof(unsigned)*8 - infvalue.precision);
  unsigned ulp_1 = (unsigned)0x01 << shift_num;
  bool carry = false;
  if(rand() <= RAND_MAX /2){
    carry = limb_add(limb,ulp_1);
  }
  if(carry){
    ++infvalue.exponent;
    limb[significand_size-1]=(~0)^((~0)>>1);
  }
  return infvalue;
}
```

Fig. 11. Infinite-precision basic stochastic function with the CESTAC strategy.

Instead of altering bits of the significand in the value, the CESTAC strategy randomly chooses different rounding mode in its basic stochastic function. Hence, every operation uses different rounding mode that yields stochastic results. Fig. 11 presents the stochastic rounding switching for infinite-precision values, which determines the ulp (unit in the last place) by the precision in the `type_real`.

In our toolchain, the tools `fpstoc` and `ipstoc` instrument the fixed-precision and the infinite-precision floating point program with the stochastic transformations, respectively. Table 1 presents the transformation rules that substitute values in numerical programs with the basic stochastic functions `p_fix` (p_{fix}) and `p_inf` (p_{inf}) defined from Figs. 8, 9, 10, and 11. According to the rules, our toolchain automatically transforms the operations in each numerical expression e to a stochastic form $p(e)$ when e satisfies the conditions in Table 1. The stochastic program calls the basic stochastic functions p_{fix} and p_{inf} in computing its result. The tool `fpstoc` instruments p_{fix} while `ipstoc` instruments p_{inf}. The principle in applying the basic stochastic functions is that if any values of infinite precision occur in the expression, we use the function p_{inf}, otherwise we use p_{fix}. In Table 1, the transformation handles function call $f(e)$, unary negative operator $-$, and different kinds of binary operators $\odot$, such as $+$, $-$, $\times$ and $\div$. After recursively applying the transformation on each numerical expression in the program, all numerical values in a program are perturbed systematically by the basic stochastic function(s).

Algorithm 4 describes the testing algorithm leveraging the stochastic transformation. It scans over all numerical expressions in the program $\mathcal{P}$, and replaces each numerical expression e to its stochastic form $p(e)$. When we test the program, we automatically apply the perturbed input $p_{\text{fix}}(i)$ and $p_{\text{inf}}(j)$, and output the perturbed results.

TABLE 1
Stochastic Transformation Rules

condition	$p(e)$
$e = v_{\text{fix}}$	$p_{\text{fix}}(v)$
$e = v_{\text{inf}}$	$p_{\text{inf}}(v)$
$f : F \rightarrow F \wedge e = v_{\text{fix}}$	$p_{\text{fix}}(f(e))$
$f : R \rightarrow R \wedge e = v_{\text{inf}}$	$p_{\text{inf}}(f(e))$
$e = -e_1 \ \wedge e_1 = v_{\text{fix}}$	$-p_{\text{fix}}(e_1)$
$e = -e_1 \ \wedge e_1 = v_{\text{inf}}$	$-p_{\text{inf}}(e_1)$
$e = e_1 \odot e_2 \wedge e_1 = v_{\text{fix}} \wedge e_2 = v_{\text{fix}}$	$p_{\text{fix}}(p_{\text{fix}}(e_1) \odot p_{\text{fix}}(e_2))$
$e = e_1 \odot e_2 \wedge e_1 = v_{\text{inf}} \wedge e_2 = v_{\text{inf}}$	$p_{\text{inf}}(p_{\text{inf}}(e_1) \odot p_{\text{inf}}(e_2))$

Algorithm 4. Stochastic Testing Algorithm

input: $\mathcal{P}$ // the program to be tested
input: I // a set of inputs
1: $\forall\, e \in \mathcal{P}, i, j \in I \wedge i \in \mathscr{I} \wedge j \in \mathscr{I}$ **do**
2: output $\mathcal{P}[e \mapsto p(e)](p_{\text{fix}}(i), p_{\text{inf}}(j))$
3: **end for**

3.4 `ediagno`: Evaluation and Diagnosis

`Ediagno` is a test harness that drives the evaluation of the transformed programs. When running a stochastic program n times, we get a set of outputs $\{o_t | 1 \le t \le n\}$. Then `ediagno` detects and diagnoses numerical instabilities by the statistical properties of the outputs, then it generates fixing hints when the instabilities are caused by practice.

The tool needs to measure if the outputs vary a lot. Root mean square error (RMSE), also called the root mean square deviation, is a frequently-used measurement which is originally designed for analyzing the relative errors for values in statistics [11]. It is expressed as the following formula in our toolchain

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(o_t - o)^2}, \tag{5}$$

where o_t is one of the stochastic outputs, o is the corresponding non-perturbed output, and n is the stochastic running times. Although a larger n increases accuracy in statistics, it costs significantly more computing resources in testing when we need to compute infinite-precision values in every execution. Hence, choosing a practical n in testing is a tradeoff. Later in our evaluation, we discuss more about the choice of n.

When we compare the outputs of different subjects, we normalize the RMSE measurements by the coefficient of variation, which is denoted as CV(RMSE)

$$\text{CV(RMSE)} = \frac{RMSE}{\mu} = \frac{\sqrt{\frac{1}{n}\sum_{t=1}^{n}(o_t - o)^2}}{\frac{1}{n}\sum_{t=1}^{n} o_t}, \tag{6}$$

where μ is the mean of stochastic outputs.

Although CV(RMSE) measures the variation of stochastic outputs, our toolchain does not directly use it, because we need to study how the output variation changes with the inputs. For this reason, we introduce two measurements for fixed-precision stochastic evaluation and infinite-precision stochastic evaluation, respectively. We call them the implementational condition number and the statistical condition number

$$\text{ICN} = \text{CV(RMSE)}_{\text{fix}}/m_{\text{fix}} \tag{7}$$

$$\text{SCN} = \text{CV(RMSE)}_{\text{inf}}/m_{\text{inf}}. \tag{8}$$

Both the measurements use the corresponding CV (RMSE) and divide it by the perturbation magnitude m, which is shown in Equation (4). Hence, they provide the ratio of the output variation over the input variation. A large implementational condition number (or statistical condition number) means that a small change of an input or an intermediate value causes the outputs fluctuate a lot. In the program-level, a large implementational condition number suggests the program is unstable, while in the problem level, a large statistical condition number shows that the problem defined in the requirements is unstable by its nature.

As a tradition of numerical analysis, condition number (CN, referred as theoretical condition number) is a well known measurement that shows the sensitivity of the output by the input of the numerical problem [12]. Here we design the ICN and SCN based on the concept of CN, yet we make the ICN and SCN more practical. The theoretical condition number (CN) is defined as the ratio of the output relative error to the input relative error. When we compute CN of a function $y = f(x)$, the formula is:

$$\text{CN} = \left|\frac{dy/y}{dx/x}\right| = \left|\frac{dy}{dx} * \frac{x}{y}\right| = \left|f'(x) * \frac{x}{f(x)}\right|. \tag{9}$$

Since we target on practicality, the design of ICN and SCN is different from the theoretical condition number. Because of the round-off errors, computable real numbers often lose precision when they are represented as fixed-precision floating point numbers. Large ICNs may also be caused by improper implementations of the numerical program. Hence, it is insufficient to use ICN to indicate the properties of the mathematical problem. Instead, we use SCN on the infinite-precision program. SCN provides insight about problem stability similar to CN. But the two are practically different because CN provides the ratio when the relative error becomes infinitesimal. For example, when we discuss the function $y = \sin(x)/x$, the theoretical condition number is:

$$\text{CN} = \left|\frac{dy}{dx} * \frac{x}{\sin(x)/x}\right| = \left|\frac{x\cos(x)}{\sin(x)} - 1\right|. \tag{10}$$

According to the CN values, $y = \sin(x)/x$ is unstable when $x = \pi * 10^{10}$, and stable when $x = \pi * (10^{10} + 1/2)$, which is strange as numerical programs should have similar precision when computing the function with both inputs. In contrast, the SCN values tell us the function actually accumulates similar errors when we precisely compute it with $x = \pi * 10^{10}$ and $x = \pi * (10^{10} + 1/2)$ with a computer, even with infinite precision arithmetic. Hence, SCN has more practical value for problems implemented by real world programs.

In spite of this, some rules regarding CN are still applicable to ICN and SCN. One example is the threshold for instability detection and diagnosis. Several numerical analysis approaches use a typical value of 10 as the threshold of CN [13]. Others in numerical analysis suggest that when CN is greater than 10, the result at least loses one decimal digit of precision in the significand [12]. Empirically, the same threshold 10 works well in our evaluation too. Hence our toolchain uses 10 as its default threshold for ICN and SCN.

The tool `ediagno` computes ICN and SCN from the outputs of the stochastic programs. Then it uses the threshold 10 from the traditional practice to detect instabilities. When ICN is larger than the threshold, the numerical program is unstable. Then it further checks the SCN. When it is also larger than the threshold, it diagnoses the instabilities are caused by problem, and suggests the user to inspect the

requirements. Otherwise, the instabilities are caused by practice. In this case, users can start an optional post-analysis in `ediagno`, which provides hints to help developers fix the numerical implementation.

3.5 Optional Post-Analysis

The optional post-analysis of our tool-chain provides localization and fixing hints for numerical instabilities caused by practice when ICN is large. The analysis collects traces of two groups of executions, one from the fixed-precision stochastic program and the other from the corresponding infinite-precision program. Then it compares them to identify salient differences that indicate the location of the numerical instability in the program.

When users start the optional post-analysis, our tool-chain links a tracing version of the basic stochastic functions `p_fix'` and `p_inf'` to the transformed programs instead of `p_fix` and `p_inf` defined in Figs. 8, 9, 10, and 11. The function `p_fix'` not only performs stochastic perturbation as `p_fix`, but also records its input along with an expression index when it is called. Hence it traces every intermediate value in an execution of the fixed-precision stochastic program. Meanwhile, the function `p_inf'` records the infinite-precision value of the corresponding expression without any stochastic perturbation, which provides a testing oracle of every intermediate value in the program.

With the new version of stochastic functions, an execution of the transformed infinite-precision program outputs an oracle trace $T = \langle t_1, t_2, t_3, \ldots, t_m \rangle$, whereas an execution of the fixed-precision stochastic program records a test trace $\hat{T} = \langle \hat{t}_1, \hat{t}_2, \hat{t}_3, \ldots, \hat{t}_m \rangle$. We define every intermediate tracing value t_i (or $\hat{t}_i$)$(1 \leq i \leq m)$ as a tuple (e, v) , where e is the recorded expression label and v is the corresponding recorded value. When the post-analysis executes the fixed-precision stochastic program n times, we get a set of test traces $\hat{\mathcal{T}} = \{\hat{T}_1, \hat{T}_2, \ldots, \hat{T}_n\}$.

Algorithm 5 defines the algorithm of localization by analyzing traces. The algorithm accepts an oracle trace and a set of test traces with the same test inputs. Then it outputs the label of the first localized unstable expression in the numerical program. If the stochastic testing causes the program to execute a different path (line 2), it means errors in numerical values may also lead the program to execute a different path. Since it is often dangerous, the algorithm directly returns the first different expression in the path to warn users (line 3). Then the algorithm computes ICN for every intermediate value in the trace. According to Equation (7),

$$\mathrm{ICN} = \frac{\mathrm{CV(RMSE)}_{\mathrm{fix}}}{m_{\mathrm{fix}}} = \frac{\sqrt{\frac{1}{n}\sum_{t=1}^{n}(o_t - o)^2}}{m_{\mathrm{fix}} \times \frac{1}{n}\sum_{t=1}^{n} o_t}. \tag{11}$$

Here we substitute o_t with every intermediate value $\hat{t}_i.v$ in test traces, and substitute o with $t_i.v$ in the oracle trace (lines 5-6). Finally the algorithm outputs the expression label whose ICN is greater than the threshold 10 discussed in Section 3.4 as the potential reasons of implementational instabilities.

The optional post-analysis also outputs an expression which is mathematically equivalent to the localized numerical expression as a fixing hint. It transforms the localized expression with the commutative, associative, and distributive laws, and outputs an expression whose fixed-precision floating-point result is the nearest one to the value in the oracle trace. Details of the expression transformation can be further referred to one of our previous work [14].

Algorithm 5. Localization Algorithm

input: T // an oracle trace
$\hat{\mathcal{T}}$ // the corresponding set of test traces
output: e // a label of the localized expression
1: **for** $i = 1$ **to** n **do**
2: **if** $\exists \hat{T}_j \in \hat{\mathcal{T}}, \hat{t}_i = \hat{T}_j.\mathrm{value}(i), t_i = T.\mathrm{value}(i), \hat{t}_i.e \neq t_i.e$ **then**
3: **return** $t_i.e$
4: **end if**
5: $O_i \leftarrow \{\hat{t}_i.v | \forall \hat{T}_j \in \hat{\mathcal{T}}, \hat{t}_i = \hat{T}_j.\mathrm{value}(i)\}$
6: $\mathrm{icn}_i \leftarrow \mathrm{ICN}(O_i, t_i.v)$; // compute ICN
7: **if** $\mathrm{icn}_i > 10$ **then**
8: **return** $t_i.e$
9: **end if**
10: **end for**

A real world numerical software may contain instabilities that are caused by very complicated mechanism in practice. Our post-analysis does not guarantee the localization and fixing information is complete in all scenarios. For example, a numerical instability may be caused by multiple locations in a program and our localization algorithm always outputs the first location. Meanwhile, the fixing hints take effect only when the numerical instability can be fixed by changing one numerical expression with the commutative, associative, and distributive laws. Hence, users need to further validate the outputs of our tool chain manually to fix numerical instabilities. Nonetheless, we still believe that the post-analysis is helpful especially when users detect numerical instabilities in large software. It provides locations and clues for fixing instabilities caused by practice, so users do not need to examine the whole numerical program.

4 Evaluation

We first evaluate our approach on a few subject programs from the literature, including two well known ill-conditioned problems, two are well-known problematic numerical implementations, and two stable numerical programs. By testing on these programs, we want to answer the following key questions:

RQ1: Can our approach distinguish ill-conditioned problems from unstable implementation?
RQ2: Are there general guidelines in setting parameters, such as the perturbation magnitude m? Are the commonly used thresholds from the practice of numerical analysis applicable in our context?
RQ3: Is the performance of our system reasonable?

Then we test our framework over the API functions in GNU scientific library (GSL) version 1.16 (i.e., the latest version). This is to evaluate the effectiveness of our approach on applications of industrial-strenth, aiming to answer the following research question:

RQ4: Can our toolchain detect real bugs in numerical software, and help fixing the detected bugs?

```
1. double re=0.0;
2. int n=70712;
3. for(i=1;i<n;++i) {
4.     re+=1/((double)i*i);}
```

(a) harmonic.c

```
1. double a, b, c, re;
2. a=7; b=8686; c=2;
3. re=(-b+sqrt(b*b-4*a*c))
4.                 /(2*a);
```

(b) root.c

```
1. double xi, xsi, A;
2. xi = 1;
3. A=random(20.0,30.0);
4. xsi = 2*xi-A*xi*xi;
```

(c) inv.c

```
1. double alpha,NA,re,Z,L,Lp,A;
2. Z = 4; A = 26; re = 11.3;
3. alpha = 0.7; NA = 12.5;
4. L=log(184.15/pow(Z,1.0/3));
5. Lp=log( 1194.0 /
6.           pow(Z, 2.0/3) );
7. double X0=(4.0*alpha*re*re);
```

(d) sample_run.c

```
1. vector b; MATRIX A=HMatrix;
2. for(i=0;i<rows;++i) {
3.   for(j=i+1;j<cols;++j) {
4.     b(j)=b(j)-
5.         A(j,i)*b(i)/A(i,i);
6.     for(k=i; k<cols; ++k){
7.       A(j,k)=A(j,k)-A(j,i)*
8.           A(i,k)/A(i,i);}}}
```

(e) Hilbert.c

```
1. int a=0; int b=0;
2. E = MatrixSolve(input);
3. for(j=0; j<n-1; ++j){
4.   a += Y[j] * E[0][j];
5.   b += Y[j] * E[1][j]; }
6. for(i=1; i<n; ++i)
7.   M[i] = ( input[0]-b/a )*
8.     exp(-a*i)*(1-exp(a));
```

(f) GM.c

Fig. 12. Key code segments of the evaluation subjects from the literature.

Our evaluation was performed on an Apple MacBook Pro with Intel Core i5 CPU 2.4 GHz, and 4 GB Physical Memory. The operating system is MacOSX 10.7 Lion. Our implementation is based on the ROSE compiler 0.9.5a [15] and iRRAM 2013.01 [10]. We reuse the code of exception pattern in iRRAM. When users apply the CESTAC strategy, our implementation calls CADNA and SAM library to mutate the ulp of numerical values, which are based on CADNA 1.1.9 [1] and SAM 2013 [16]. We compile the program with llvm-gcc 4.2.1.

4.1 Results for Programs from the Literature

Fig. 12 presents the salient code segments of our first set of subject programs. We collect our first set of evaluation subjects from several existing projects. The set includes two well-known ill-conditioned problems (`Hilbert.c` [17] and `GM.c` [18]), two well-known unstable numerical implementations (`harmonic.c` [19] and `root.c` [20]), and two stable implementations (`inv.c` [21] and `sample_run.c` [22]). These subjects contain 869 lines of code (LOC) in total.

The code snippet of `Hilbert.c` is shown in Fig. 12e. It represents a well-known ill-conditioned problem—the Hilbert linear equation system [17]. In particular, a Hilbert linear equation system is denoted as

$$Hx = h, \tag{12}$$

with the $n \times n$ Hilbert coefficient matrix H expressed as

$$H_{i,j} = \frac{1.0}{i + j - 1.0}, \tag{13}$$

$x_{n\times 1}$ denoting a vector containing n unknowns, $h_{n\times 1}$ the vector of constant terms.

The other well-known ill-conditioned problem is the Grey Prediction Control Model GM(1,1) [18] in Fig. 12f, which is expressed as the following differential equation

$$\frac{dX^{(1)}}{dt} + aX^{(1)} = u, \tag{14}$$

where $X^{(k)}$ is a sequence of time series numerical data, a and u are unknown coefficients that need to be estimated

$$X_i^{(k)} = \sum_{j=1}^{i} X_j^{(k-1)}. \tag{15}$$

For detecting whether solving a linear equation is ill-conditioned, traditional numerical analysis directly computes the theoretical condition number (CN) from the coefficient matrix. Unlike these traditional practices, our approach evaluates the problem directly on the system. This is very desirable when the system is complicated. In our evaluation, all the systems are transformed to the infinite-precision implementations that eliminate any possible translation errors from the mathematical models to the programs due to the precision lost in the fixed-precision arithmetic.

In addition, Figs. 12a and 12b show the code snippets for unstable implementations. In particular, the `harmonic.c` in Fig. 12a is from [19], where Stanoyevitch computes the generalized harmonic number with the order of two using a loop. When the variable `i` becomes large during the iteration, the loop continuously adds the small number $1/i^2$ onto a very large floating-point number `re`, which is an improper practice in numerical programming. The `root.c` program in Fig. 12b is from [20]. It computes the quadratic formula for some given parameters. It is unstable because there is massive cancellation in its calculations.

We also include two stable numerical programs for comparison. The `inv.c` program is from [21] and `sample_run.c` is from the computation of a mechanical engineering simulator [22]. Although some expression in

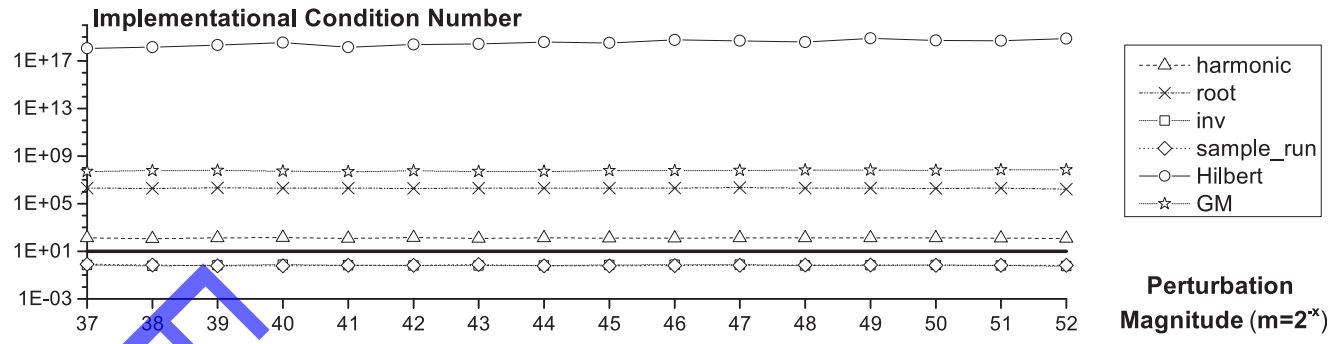


Fig. 13. ICN (implementational condition number) of the stochastic outputs over fixed-precision arithmetic with different perturbation magnitude m, which is from 2^{-52} to 2^{-37}.

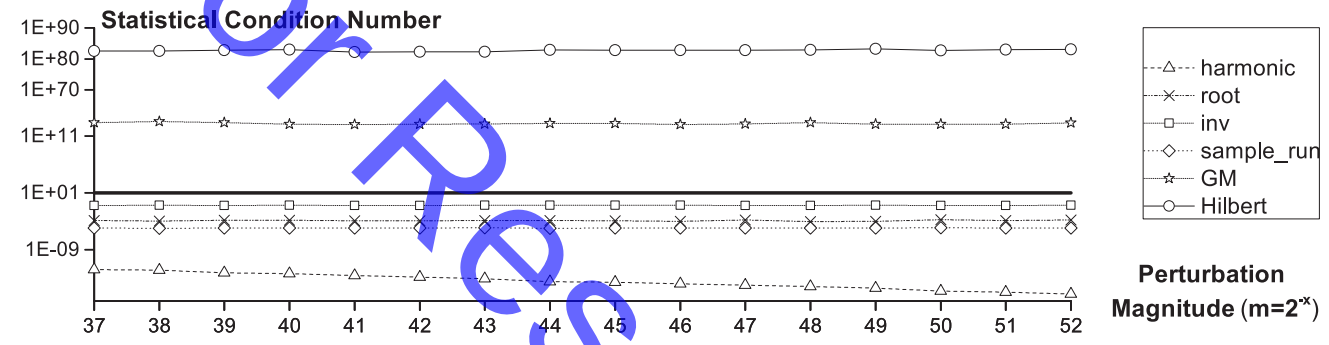


Fig. 14. SCN (statistical condition number) of the stochastic outputs over infinite-precision arithmetic with different perturbation magnitude m, which is from 2^{-52} to 2^{-37}.

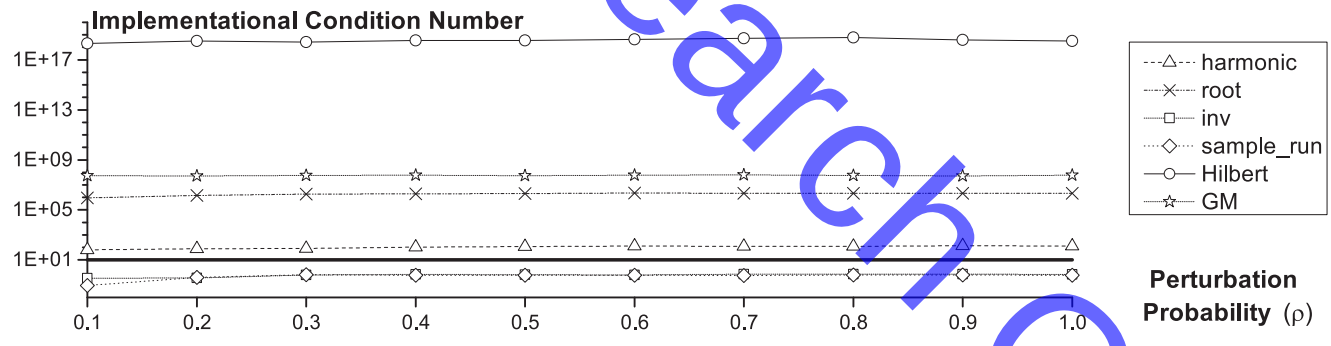


Fig. 15. ICN (implementational condition number) of the stochastic outputs over fixed-precision arithmetic with different perturbation probability ρ from 10 to 100 percent, while we fix the perturbation magnitude $m = 2^{-45}$.

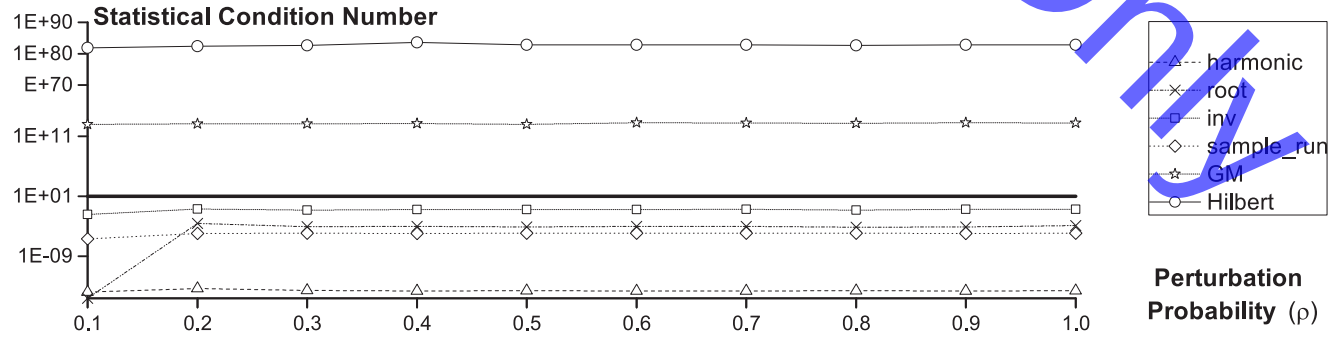


Fig. 16. SCN (statistical condition number) of the stochastic outputs over infinite-precision arithmetic with different perturbation probability ρ from 10 to 100 percent, while we fix the perturbation magnitude $m = 2^{-45}$.

the simulator is complicated, the numerical computation is correct and stable.

Figs. 13, 14, 15, and 16 and Table 2 present the results on testing the subjects from the literature. We first evaluate all the subjects with the plain stochastic strategy. Particularly, we fix the perturbation probability $\rho = 100$ percent and then systematically alter the perturbation magnitude m from 2^{-52} to 2^{-37}. We run 200 times for each magnitude value, and compute the implementational condition number and statistical condition number separately for the fixed-precision arithmetic and the infinite-precision arithmetic. Fig. 13 shows the implementational condition numbers of the fixed-precision

TABLE 2
ICN (Implementational Condition Number) and SCN (Statistical Condition Number) of the Stochastic Outputs with Different Numbers of Running Samples, While We Fix the Perturbation Magnitude $m = 2^{-45}$, and the Perturbation Probability $\rho = 50\%$

SampleNum		3	5	10	20	50	100	200
harmonic	ICN	1.482E+02	1.471E+02	1.656E+02	1.502E+02	1.038E+02	1.233E+02	1.264E+02
	SCN	4.265E-15	6.159E-16	2.586E-15	4.088E-15	2.787E-15	2.448E-15	1.917E-15
root	ICN	1.891E+06	9.056E+05	2.522E+06	2.235E+06	1.805E+06	2.117E+06	2.078E+06
	SCN	4.519E-05	6.548E-05	7.379E-05	1.292E-04	2.123E-04	1.657E-04	1.286E-04
inv	ICN	6.287E-01	6.976E-01	6.323E-01	6.527E-01	6.361E-01	6.828E-01	6.878E-01
	SCN	7.162E-02	6.262E-02	6.316E-02	7.416E-02	6.116E-02	6.716E-02	6.938E-02
sample_run	ICN	6.046E-01	6.176E-01	6.041E-01	5.573E-01	6.652E-01	6.551E-01	6.071E-01
	SCN	6.379E-06	6.279E-06	6.490E-06	6.559E-06	6.304E-06	6.462E-06	6.870E-06
Hilbert	ICN	1.861E+18	3.927E+18	3.092E+18	3.315E+18	5.443E+18	2.976E+18	3.287E+18
	SCN	3.669E+82	7.505E+82	6.447E+82	5.187E+82	9.028E+82	6.948E+82	7.187E+82
GM	ICN	8.328E+07	3.423E+07	5.345E+07	5.376E+07	4.525E+07	6.142E+07	5.851E+07
	SCN	1.729E+13	1.869E+13	1.212E+13	2.032E+13	1.820E+13	1.919E+13	1.630E+13

stochastic programs, whereas Fig. 14 presents the statistical condition numbers of the infinite-precision programs. We mark the widely accepted threshold 10 as a straight line in the figures. From the results, the unstable programs (`Hilbert`, `GM`, `harmonic` and `root`) produce larger implementational condition numbers than the stable ones (`inv` and `sample_run`), which are clearly divided by the threshold. Note that all the unstable subjects would lose more than one decimal digit of the significand than its input when the subjects produce implementational condition number larger than the threshold 10. Yet the stable subjects do not have such a problem. The results in Fig. 14 are quite different from Fig. 13. Because the instabilities in `harmonic` and `root` are from improper implementation, they produce small statistical condition numbers at the same level as the results of the stable subjects (`inv` and `sample_run`). But the statistical condition numbers of `Hilbert` and `GM` are very large since their instabilities are caused by the ill-conditioned problems. According to Figs. 13 and 14, we also observe that the implementational condition number and the statistical condition number are not sensitive to changes of the perturbation magnitude m. So we can fix the perturbation magnitude m to the median value 2^{-45} for other experiments.

The perturbation probability ρ and the number of samples affect the performance of our toolchain. Hence, we also study the valid values of these parameters. In particular, we fix the perturbation magnitude m to 2^{-45}, and use 200 samples for the stochastic approach. When we change the perturbation probability ρ from 10 to 100 percent, the corresponding implementational condition and statistical condition numbers are shown in Figs. 15 and 16, respectively. From the results, the implementational condition number and the statistical condition number become unstable when the perturbation probability ρ is lower than 10 percent, because the implementational condition number of `sample_run` and the statistical condition number of `root` deviate from the numbers with full perturbation (i.e., $\rho = 100$ percent). In general, $\rho = 30$ percent is good enough. Based on these observations, we conservatively choose $\rho = 50$ percent for other experiments to ensure the quality of the results.

We further study whether it is required for us to run each subject 200 times to get stochastic results of good quality. Chatelin [23] rigorously concludes that only two or three samples with stochastic running in numerical evaluation statistically give a confidence level of 95 percent in general. Vignes proves that the CESTAC strategy only needs three independent samples to generate valid results [7], [8]. Other numerical experts also take five samples or less as a practical way in their stochastic evaluation [24], [25]. Table 2 presents the implementational condition and statistical condition numbers under different numbers of running samples. In particular, we fix the perturbation magnitude $m = 2^{-45}$ and the perturbation probability $\rho = 50$ percent. As the results align with the observations from other numerical experts, we follow the tradition of taking five samples for further experiments, to ensure our toolchain efficiently yields good results for instability detection and diagnosis.

Table 3 presents the ICNs and SCNs generated from the plain strategies (with two different configurations) and the CESTAC strategy, whereas Table 4 compares the execution

TABLE 3
ICN and SCN Comparison between the Plain Strategies and the CESTAC Strategy
(Plain#1: $m = 2^{-45}$, $\rho = 50\%$, Five Samples; Plain#2: $m = 2^{-52}$, $\rho = 100\%$, Three Samples)

Subjects	ICN			SCN		
	Plain#1	Plain#2	CESTAC	Plain#1	Plain#2	CESTAC
harmonic	1.471E+02	9.171E+01	2.481E+04	6.159E-16	2.010E-15	2.392E-13
root	9.056E+05	2.541E+06	2.549E+04	6.548E-05	8.524E-05	2.606E-09
inv	6.976E-01	3.378E-01	1.373E+00	6.262E-02	4.200E-02	1.253E-15
sample_run	6.176E-01	6.159E-01	3.080E-01	6.279E-06	1.668E-06	9.475E-16
Hilbert	3.927E+18	5.639E+20	9.814E+15	7.505E+82	4.198E+84	8.254E+15
GM	3.423E+07	9.078E+07	2.945E+07	1.869E+13	2.042E+13	3.385E+15

TABLE 4
Execution Time (in Seconds) of the Original Program, the Testing Process with the Plain Strategies, and the CESTAC Strategy (Plain#1: $m = 2^{-45}$, $\rho = 50\%$, Five Samples; Plain#2: $m = 2^{-52}$, $\rho = 100\%$, Three Samples)

Subjects	Original	Plain#1	Plain#2	CESTAC
harmonic	7.180E-04	7.828E-02	4.869E-02	4.465E-02
root	4.900E-05	3.990E-04	2.494E-04	1.650E-04
inv	5.300E-05	2.160E-04	2.011E-04	1.840E-04
sample_run	7.300E-05	4.482E-03	2.864E-03	1.481E-03
Hilbert	5.272E-03	3.018E-01	1.849E-01	9.290E-02
GM	2.610E-04	8.949E-03	8.254E-03	7.750E-03

time of the original program and the testing process with these strategies. The configuration of Plain#1 in Tables 3 and 4 is suggested from other experiments with the perturbation magnitude $m = 2^{-45}$, the perturbation probability $\rho = 50$ percent, and five samples to generate the results. The configuration of Plain#2 is designed following the closest parameters to the CESTAC strategy with the perturbation magnitude $m = 2^{-52}$, the perturbation probability $\rho = 100$ percent, and three running samples. When the CESTAC strategy only switches the value of ulp (unit in the last place) with three samples, it runs faster than the plain strategies. But for some subjects such as `Hilbert`, it produces different ICNs and SCNs than the plain strategy because the plain strategy applies perturbation to the values that are already rounded. When the plain strategies are more sensitive to the instabilities, we use the configuration of Plain#1 for other experiments in our evaluation.

From Table 4, the toolchain introduces overhead in testing compared to the native execution time of the original program, which is about 46 times slower on average. The reason is that we introduce the infinite-precision arithmetic and the stochastic sampling. However, we still believe the performance is acceptable as a high performance testing environment can address this problem. For example, users can test and diagnose their large size numerical software on a massively parallel processing platform, and run their fixed software in a normal environment without any overhead than its original version.

According to the results in Figs. 13, 14, 15, and 16, our toolchain effectively diagnoses the two unstable subjects that are caused by improper numerical implementations. The toolchain also generates fixing hints for these subjects. It locates the expression in line 4 of Fig. 12a and the expression in line 3 of Fig. 12b. Our toolchain does not provide the correct fixing expression automatically, because `harmonic` cannot be directly patched with expression replacement, and the new (and stable) expression of `root` needs very complicated transformation that is not currently supported by our implementation. Nevertheless, it is much easier for numerical experts to fix the instabilities with the localization hints in the code, especially when the project is large.

Fig. 17 shows the code patches of the two unstable subjects. For `harmonic`, we change the iteration sequence to avoid adding small numbers onto the large floating-point number. And for `root`, we use an equivalent expression $-2c/(b + \sqrt{b^2 - 4ac})$ instead of the quadratic formula $(-b + \sqrt{b^2 - 4ac})/2a$, because

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \times \frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}} = -\frac{b^2 - (b^2 - 4ac)}{2a(b + \sqrt{b^2 - 4ac})} = \frac{-2c}{b + \sqrt{b^2 - 4ac}}.$$

After the expression substitution, `root` does not have massive cancellation in its code and becomes stable. When verifying the patched code with our toolchain, we get the implementational condition numbers in Table 5. Both subjects produce low implementational condition numbers after applying the patches, according to the data in Table 5.

4.2 GSL

Besides the experiments on the subjects from the literature, we also apply the toolchain on the GNU Scientific Library [26] to detect and diagnose instabilities in real projects. The GNU Scientific Library is a mature, well-maintained, and widely deployed numerical library in industry. It is both challenging and important for us to find instabilities in the GSL functions.

The GSL library contains 2,115 functions with 275,304 lines of code (LOC). A function in the library contains a maximum of 609 lines of code, and about 130 lines of code in average. Our toolchain analyzes programs not library functions, so we compose a driver for the GSL functions by synthesizing a `main` and calls the nine elementary functions and 160 special functions with inputs falling into the function domain. For functions that are very similar to each other, we only test one of them.

For test input generation, we follow a few empirical guidelines as proposed in the genetic algorithm by Zou et al. [27]. According to [27], many functions in GSL may produce substantially different outputs due to rounding errors or/and cancellations around a few special inputs, which can be summarzied as $I = \{-e, -\pi, -2, -1, -\pi/2, 0, \pi/2, 1, 2, \pi, e\}$. Then let $\epsilon = 5 \times 10^{-10}$, and $\forall i \in I$, we randomly select 20 inputs in the range of $(i - \epsilon, i + \epsilon)$, including i itself, where we got 220 test inputs. Then we generate 100

```
1. double re=0.0;
2. int n=70712;
3. for(i=n-1;i>0;--i) {
4.     re+=1/((double)i*i);}
```

(a) harmonic_fix.c

```
1. double a, b, c, re;
2. a=7; b=8686; c=2;
3. re=(-2*c)
4.      /(b+sqrt(b*b-4*a*c));
```

(b) root_fix.c

Fig. 17. Patched code segments for unstable subjects caused by improper numerical implementations.

TABLE 5
Implementational Condition Number of the Patched Subjects

m (2^x)	-37	-39	-41	-43	-45	-47	-49	-51
harmonic_fix	6.894E-01	7.608E-01	6.862E-01	6.588E-01	7.202E-01	7.062E-01	7.686E-01	8.069E-01
root_fix	6.565E-01	6.402E-01	6.548E-01	5.588E-01	6.456E-01	6.702E-01	6.458E-01	5.829E-01

test inputs according to the algorithms in [27], and select 100 sample inputs following uniform distribution. The total number of test cases for each function is 420.

According to Barr's research [4], many underflow and overflow exceptions are avoidable in GSL. And since our transformed infinite-precision programs can handle a larger range of numerical values, we manually fix a few overflows and underflows in GSL to enlarge the valid input ranges.

Table 6 shows the experiment setup and summary of results in the GSL evaluation. With the observation in Section 4.1, we fix the perturbation magnitude $m = 2^{-45}$ and the perturbation probability $\rho = 50$ percent. For each test, we take five samples to get the correct results. We totally find 15 unstable functions, and nine of them are diagnosed as implementational instabilities. We reported the nine functions as real bugs to the GSL Buglist[3] as Bugs #45726, #45730, and #45746. The average testing time for a function is 7.284E-2 seconds, which is acceptable in testing.

Table 7 presents the evaluation details of the 15 unstable functions. We mark the nine functions with implementational vulnerabilities with grey. Table 7 presents the input, output, implementational condition number, statistical condition number and the average testing time for each function. Some of the functions are obviously wrong: As common knowledge in mathematics, the trigonometric function `sine` should output values in the range $[-1, 1]$. However the mean output of the GSL library function `gsl_sf_sin`, which actually computes values of `sine`, is $-5.206E + 118$.

Our toolchain also reports some hints for the root cause of the implementational vulnerabilities. It can narrow down to a few expressions. Here we use a code snippet as a typical example in Fig. 18. The root cause is located at line 533 of the source file `specfunc/trig.c` in GSL-1.16.

The function `angle_restrict_symm_err()` in Fig. 18 works for periodic functions with the period of π(or 2π). It tries to extract the equivalent value `r` in $(-\pi, \pi)$ for the input `theta` of the periodic function. For example, if the user provides an input `theta` of $-72\pi - 0.6$, the function in Fig. 18 should compute `r` as -0.6, because for some periodic function like `cosine`, we have $\cos(-72\pi - 0.6) = \cos(-0.6)$. From Fig. 18, the developer carefully divides π to three double values as `P1`, `P2` and `P3` for more mantissae. Then computes `y` for the number of periods in `theta`, such as `y` = -72 when `theta` = $-72\pi - 0.6$. At line 14 of Fig. 18, it computes `r` with `theta-y * π`. Hence, it is clear that the author of GSL believes `r` $\in (-\pi, \pi)$.

The problem of the code in Fig. 18 is triggered when the input `theta` becomes large. In this case, we cannot compute correct `y` through the expression in line 12, because `fabs(theta)/TwoPi` has very large absolute errors (> 1). Although the relative error of `y` is quite small, the massive cancellation in line 14 causes severe errors in `r`. When we set the input `theta` = $-7.294E + 23$, the code outputs `r` = 5092122.317. It violates the rule `r` $\in (-\pi, \pi)$ and causes problems in later computing. As the massive cancellation at line 14 is unavoidable, we suggest the developer to increase the precision of each value in computing the expression at 12. We have reported all the bugs and the patch suggestions to the Buglist of GSL.

4.3 Limits and Summary

The testing technique is not sound—it may miss important inputs (such as the GSL test inputs of unstable functions). Although infinite-precision arithmetic is precise when the computer has enough resources, it may still suffer from the precision loss when the physical computing resources is limited. Nonetheless, we are still confident that our toolchain is useful to numerical code developers. Software testing has been proven to be a useful technique for improving software quality. The test suite is representative and the reported results are promising. Our toolchain did find real bugs in the latest version of one of the most well-maintained, widely deployed numerical libraries in industry.

Diagnosing whether a numerical instability is caused by the implementation is important for users, since it tells them if it is a bug that can be fixed by changing the code. Otherwise, the users may need to re-formulate their problem. From the results of our evaluation, we have several observations: For *RQ1*, the numerical instabilities from ill-conditioned problems generate large implementational condition numbers as well as statistical condition numbers. But unstable implementations only generate large implementational condition numbers with small statistical condition numbers. Hence, they can be easily distinguished by our toolchain. For *RQ2*, the results are not sensitive to the perturbation magnitude m. A median value of $m = 2^{-45}$ is recommended in general. To ensure result validity, we recommend the perturbation probability $\rho \geq 30$ percent and the number of samples ≥ 3. For distinguishing and diagnosis instabilities in numerical programs, the traditional practice suggested threshold used on CN is also applicable on

TABLE 6
GSL Testing Parameters and Summary

No. of Tested Functions	169
No. of Inputs	420
Perturbation Magnitude m	2^{-45}
Perturbation Probability ρ	50%
No. of Perturbation Samples	5
No. of Fixed Overflows	57
No. of Fixed Underflows	103
No. of Detected Instabilities	15
No. of Detected Implementation Errors	9
Average Testing Time (sec.)	7.284E-2

3. https://savannah.gnu.org/bugs/?group=gsl

TABLE 7
The 15 Unstable Functions Detected in GSL

Function	Input	Mean Output	ICN	SCN	Average Time (sec.)
gsl_log1p	-9.999E-01	-1.868E+01	2.069E+06	2.675E+05	1.322E-01
gsl_sf_bessel_I0	-7.062E+02	7.684E+304	6.715E+01	1.788E+02	5.184E-03
gsl_sf_bessel_I1	6.324E+02	7.373E+272	7.976E+01	1.211E+02	5.643E-03
gsl_sf_bessel_j0	-1.458E+27	6.686E+152	1.756E+15	1.742E+00	6.454E-02
gsl_sf_bessel_j1	-7.013E+25	-1.367E+140	1.401E+14	6.258E-01	7.761E-02
gsl_sf_bessel_j2	8.933E+21	1.112E+89	3.710E+14	4.359E-01	4.120E-02
gsl_sf_bessel_y0	9.372E+22	2.167E+102	2.304E+14	5.629E-01	4.713E-02
gsl_sf_bessel_y1	6.496E+23	3.030E+111	1.074E+16	5.105E-01	5.600E-02
gsl_sf_bessel_y2	4.448E+28	-4.170E+175	1.270E+14	1.108E+00	1.108E-01
gsl_sf_clausen	2.291E+14	-1.407E-02	1.878E+15	3.221E+00	4.630E-02
gsl_sf_cos	-7.294E+23	-7.317E+136	1.726E+14	1.657E+00	2.258E-02
gsl_sf_eta	-1.700E+02	6.838E+220	3.922E+02	2.234E+03	9.675E-03
gsl_sf_gamma	1.710E+02	5.738E+306	3.775E+02	3.781E+03	9.837E-03
gsl_sf_psi	-9.999E-01	-1.556E+14	7.751E+13	1.329E+11	9.162E-03
gsl_sf_sin	3.548E+22	-5.206E+118	4.902E+14	3.597E+00	2.415E-02

the measurement of ICN and SCN in our evaluation. The toolchain diagnoses the subjects from the literature with perfect accuracy, and detects real bugs in GSL with the suggested threshold. For *RQ3*, our toolchain introduces overhead in numerical testing due to the infinite-precision numerical computing, stochastic sampling etc. Hence, the process of testing is about 46 times slower than the native execution of the original program. Nevertheless, we believe the performance is still acceptable because a high performance testing environment can address this problem. For example, users can test and diagnose their large size numerical software on a massively parallel processing platform, and run their fixed software in a normal environment without any overhead than its original version. And for *RQ4*, our toolchain finds real bugs in the latest version of GSL.

5 Related Work

Computing the propagation of roundoff errors and analyzing (in)stability of numerical algorithms is a popular research topic that has been studied for more than two decades. A great number of theoretical approaches have been proposed to help engineers manually analyze numerical problems and develop stable programs [28], [29], [30], [31], [32]. This section mainly surveys a few recent practices on automatic techniques for improving numerical stability. Five threads are included from different aspects: 1) static analysis of numerical software based on the interval arithmetic and affine arithmetic, 2) dynamic testing and test generation to detect numerical instabilities, 3) verification of numerical properties and program transformation for numerical optimization, 4) the development of infinite-precision arithmetic, and 5) perturbation techniques and the stochastic arithmetic.

```
1  int gsl_sf_angle_restrict_symm_err_e
2      (const double theta, ... )
3  {
4    const double P1 = 4 * \
5          7.85398125648498535156250e-01;
6    const double P2 = 4 * \
7          3.77489470793079817667600e-08;
8    const double P3 = 4 * \
9          2.69515142907905948405520e-15;
10   const double TwoPi = 2*(P1 + P2 + P3);
11
12   const double y = GSL_SIGN(theta) * 2 * \
13         floor(fabs(theta)/TwoPi);
14   double r = ((theta - y*P1) -y*P2) -y*P3;
15   ... ...
16 }
```

Fig. 18. A code snippet that causes errors in GSL.

Static Analysis for Numerical Software. A few static approaches are proposed to analyze the possible numerical errors based on the theory of interval arithmetic [33] or affine arithmetic [34]. Interval arithmetic uses an error range, also called *interval*, to represent a numerical value, and substitutes the basic operations on numbers as the operations between intervals. For example, instead of using the number of 2.0, interval arithmetic uses a range [1.97, 2.03] that means we are sure that the value must be somewhere between 1.97 and 2.03. Then it substitutes adding two numbers with the operations of adding two intervals that outputs the maximum possible range, as $[a, b] + [c, d] = [a + b, c + d]$. Affine arithmetic further tracks the sources of errors as affine forms for better precision. It presents the numerical value as the affine form of $x_0 + x_1\epsilon_1 + x_2\epsilon_2 + x_3\epsilon_3 ...$, where x_0 is the central value of the form, often representing the precise value, x_i are the partial deviations often represents errors coming from different sources, and ϵ_i are the noise symbols whose values are unknown but assumed to be in the interval $[-1, 1]$. Besides recording an error range as interval arithmetic, affine arithmetic also tracks the correlations between sources. Hence, affine arithmetic yields a tighter range than the interval arithmetic when the value contains errors from the same source. Several static approaches work with the interval and affine arithmetic to estimate the propagation of errors in numerical programs. Putot et al. propose a static analysis on C code, which relies on abstract interpretation by interval values and arithmetic [35]. Goubault [36] gives a semantics of floating-point operations and abstract them with the affine forms to extract information about the possible loss of precision. Later Goubault and Putot [21] refine their work with an advanced abstract domain of affine arithmetic, which highly increases the precision of such analysis.

Numerical Testing and Test Case Generation. Dynamic testing for improving numerical stability is another well

studied topic in recent years. Benz et al. [37] present a tool called FPDebug, which uses a dynamic testing based on binary translation and lightweight program slicing to detect numerical bugs in programs. Bao and Zhang [38] further reduce the cost of detection by not computing all the precise errors but only tracking several potentially inaccurate values. For test generation, Miller and Spooner [39] propose a search-based technique, which they called numerical maximization method to get floating-point input data. Bagnara et al. [40] use several search heuristics to improve constraint-solving in concolic testing for floating-point programs. Barr et al. [4] work on a symbolic execution approach that finds numerical exceptions such as overflows and underflows in software. Recently, Zou et al. [27] improve the tool FPDebug with a genetic algorithm, which automatically generates test inputs that aim to trigger significant inaccuracies in floating-point programs.

Verification and Transformation. Some recent researches also focus on verifying floating-point properties and transforming numerical programs for different purpose. Boldo et al. [41], [42] introduce support systems to verify the properties of floating-point C programs with the Coq proof assistant. Ayad and Marché [43] propose an expressive language to formally specify floating-point properties, and use multiple provers to generate proofs of such properties. Darulova and Kuncak [44] present a type system for rigorous reasoning the precision of floating-point results. Numerical program transformation is also well studied recently. For precision optimization, Martel [45] proposes a concrete semantics to generally explain the propagation of rounding errors. He implements an optimizer [2] to statically transform numerical programs to more precise ones in his later works [3]. Monniaux [46] points out a few pitfalls in semantics of floating-point numbers such as subtle differences in hardware platforms and compilers can pose special challenges in verifying floating-point computation. Monniaux further proposes some alterations to program verification techniques to solve the problem. For performance optimization, Lam et al. [47], Rubio-González et al. [48], and Schkufza et al. [49] separately propose different precision tuning and adapting approach.

Arbitrary and Infinite Precision Arithmetic. Arbitrary precision arithmetic (APA) and infinite precision arithmetic, are recently developed techniques for computing real numbers numerically without errors [50]. The APA technique uses dynamic allocated memories to represent floating-point numbers in a computer, which indicates that the digits of precision are limited only by the available memory of the system. Marcial-Romero and Escardo propose the semantics of APA in programming language, and discuss the logic properties of such semantics [51], then Ménissier-Morain theoretically gives the operators and algorithms of APA [52]. Intervals analysis based on APA is discussed by Blanck [53], and implemented in [54]. Anderson et al. compare the performance of different implementations that work on decimal floating-point numbers [55]. And a few researches focus on the hardware configuration that improves the efficiency of APA and IPA. Wang et al. define the floating-point division, floating-point square root and floating-point accumulation for the reconfigurable hardware [56], then they extend their work to a tool called VFloat [57], which supports full features for APA under reconfigurable hardware. El-Araby et al. discuss the hardware features such as the virtual convolution scheduling and the dynamic pipeline in improving the performance of APA and IPA [58]. A few libraries support the computation of APA and IPA, which include the GNU multiple precision arithmetic library (GMP) [59], the GNU multiple precision floating-point reliable library (MPFR) [9], the e_float Library [60] and the number theory library (NTL) [61] for APA; And the RealLib [62] and iRRAM [10] for IPA. In this paper, we implement the IPA based on iRRAM by Norbert Müller [10], which is the fastest state-of-art IPA package [62].

Perturbation and Stochastic Arithmetic. The stochastic arithmetic works based on the Monte Carlo approach by randomizing floating-point computation. The pioneer work on numerical stochastic arithmetic is CESTAC [7], [63], which consists in executing each floating-point operation several times with a random rounding mode. Discrete Stochastic Arithmetic [8] is based on the CESTAC method and is implemented in the CADNA [1] library. With the hypothesis that the elementary round-off errors of the floating-point arithmetic are randomly independent, centered and uniformly distributed variables, CADNA only needs about three different executions to conclude the stability of the programs. Parker et al. [20], [64] formalize the Monte Carlo Arithmetic (MCA) framework, which is similar to the basic insight of stochastic arithmetic. This framework allows both random rounding that simulates uncorrelated roundoff errors, and random unrounding that simulates catastrophic cancellation with errors correlated to the numerical operations. Based on MCA, Eggert and Parker [24] develop a tool called wonglediff, which changes the rounding mode of FPU to evaluate the numerical programs at runtime. It is restricted to rounding modes perturbation with unmodified numeric programs. Tang et al. [14] introduce expression perturbation to localize the instabilities in numerical implementations. Different from these tools, our toolchain combines the stochastic arithmetic over both fixed-precision and infinite-precision floating-point numbers and uncovers numerical instabilities from software requirements.

6 Conclusion and Future Work

Analyzing (in)stability of numerical computing is a key process for building reliable software. In this paper, we have presented the design and implementation of a numerical analysis toolchain that not only detects the potential instability in software, but also diagnoses reason for such instability. We classify the reason of instability into two categories: When it is introduced by the software requirement specification, we say the instability is caused by problem. In this case, developers should think about improving their software at a higher level, which means redefining the software requirements to mitigate or get around the instability inducing mathematical properties. Otherwise, we say the instability is caused by practice and developers can improve the software itself with better numerical design and programming skills. We build our toolchain with four loosely-coupled tools: `ipatrans` transfers the numerical program to an infinite-precision arithmetic form, which dynamically iterates the numerical computation with more precise numbers to uncover the distance between the

program implementation and the mathematical insight behind it. Then `fpstoc` and `ipstoc` separately apply the stochastic perturbation on both the fixed-precision program and the infinite-precision form. Finally, `ediagno` harness the testing, which statistically synthesizes the results to detect the potential instabilities and diagnoses the reasons for such instabilities—by the mathematical properties of the problem or by improper practices in the implementation. We evaluate a few subjects from the literature and functions in the GNU Scientific Library. From our evaluation, we get several interesting conclusions: 1) The numerical instabilities from ill-conditioned problems generate large implementational condition numbers as well as large statistical condition numbers. However, unstable implementations only generate large implementational condition numbers with small statistical condition numbers. So they can be easily distinguished by our toolchain. 2) The ICN and SCN are not sensitive to the perturbation magnitude m. A median value of $m = 2^{-45}$ is recommended in general. To ensure result validity, we recommend the perturbation probability $\rho \geq 30$ percent and the number of samples ≥ 3. The toolchain diagnoses the subjects from the literature with perfect accuracy, and detects real bugs in GSL with the suggested threshold. 3) The performance of our toolchain is quite acceptable. The average time for testing a function in the numerical library in industry is only 7.284E-2 second on an ordinary laptop. And after testing, the numerical code can be run without any overhead than its original version. 4) Our toolchain finds real bugs in the latest version of GSL. We report fixing advices to its bug list.

In future, we will develop more features in the toolchain, such as improve the post-analysis to detect patterns of benign path changing and integrate modules that automatically find critical inputs in numerical testing. And we are also going to improve the interface of our toolchain to fit more complicated testing, including the industry software that processes multimedia contents. Furthermore, we would improve the framework with parallel and distributed techniques to fill the gap between theoretical research activities and practical engineering applications.

Acknowledgments

We would like to thank Xie Wang and Shixun Huang for their helpful discussions. This research is supported by National Basic Research Program of China (973 Program 2014CB340702), National Natural Science Foundation of China (Grant No. 61402222, 61632015 and 61373013), and NSF(US) Award (Grant No. 1409668, 1320444, and 1320306). Enyi Tang and Zhenyu Chen are the corresponding authors.

References

[1] F. Jézéquel and J. M. Chesneaux, "CADNA: A library for estimating round-off error propagation," *Comput. Physics Commun.* vol. 178, no. 12, pp. 933–955, Jun. 2008, doi: 10.1016/j.cpc.2008.02.003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465508000775

[2] M. Martel, "Semantics-based transformation of arithmetic expressions," in *Proc. 14th Int. Conf. Static Anal.*, 2007, pp. 298–314.

[3] M. Martel, "Program transformation for numerical precision," *Proc. ACM SIGPLAN Workshop Partial Eval. Program Manipulation*, 2009, pp. 101–110, doi: 10.1145/1480945.1480960.

[4] E. T. Barr, T. Vo, V. Le, and Z. Su, "Automatic detection of floating-point exceptions," *Proc. 40th Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 2013, pp. 549–560, doi: 10.1145/2429069.2429133.

[5] N. Mohan, T. M. Undeland, and W. P. Robbins, *Power Electronics: Converters, Applications, and Design*, 3rd ed. Hoboken, NJ, USA: Wiley, 2002.

[6] D. W. Hart, *Power Electronics*, 1st ed. New York, NY, USA: McGraw-Hill Education, 2010.

[7] J. Vignes, "A stochastic arithmetic for reliable scientific computation," *Math. Comput. Simul.*, vol. 35, no. 3, pp. 233–261, 1993.

[8] J. Vignes, "Discrete stochastic arithmetic for validating results of numerical software," *Numerical Algorithms*, vol. 37, no. 1–4, pp. 377–390, Dec. 2004, doi: 10.1023/B:NUMA.0000049483.75679.ce. [Online]. Available: http://link.springer.com/article/10.1023/B\%3ANUMA.0000049483.75679.ce

[9] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, Jun. 2007, Art. no. 13, doi: 10.1145/1236463.1236468.

[10] N. Müller, "The iRRAM: Exact arithmetic in C++," in *Computability Complexity Analysis*, Blanck J, Brattka V, Hertling P, Eds., Berlin, Germany: Springer, 2001, pp. 222–252.

[11] J. Cort and K. M. Willmott, "Advantages of the mean absolute error mae over the root mean square error RMSE in assessing average model performance," *Climate Res.*, vol. 30, pp. 79–82, 2005.

[12] E. Cheney and D. Kincaid, "Numerical mathematics and computing," International student edition, Cengage Learning, 2007. [Online]. Available: https://books.google.com/books?id=ZUfVZELlrMEC

[13] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken, *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. Taylor & Francis, 2013. [Online]. Available: https://books.google.com/books?id=gkalyqTMXNEC

[14] E. Tang, E. Barr, X. Li, and Z. Su, "Perturbing numerical calculations for statistical analysis of floating-point program (in) stability," *Proc. 19th Int. Symp. Softw. Testing Anal.*, 2010, pp. 131–142, doi: 10.1145/1831708.1831724.

[15] D. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel Process. Lett.*, vol. 10, pp. 215–226, 2000.

[16] S. Graillat, F. Jézéquel, S. Wang, Y. Zhu, "Stochastic arithmetic in multiprecision," *Math. Comput. Sci.*, vol. 5, no. 4, pp. 359–375, Nov. 2011, doi: 10.1007/s11786-011-0103-4. [Online]. Available: http://link.springer.com/article/10.1007/s11786-011-0103-4

[17] C. Liu and C. Chang, "Novel methods for solving severely ill-posed linear equations system," *J. Marine Sci. Technol.*, vol. 17, pp. 216–227, 2009.

[18] X. Xiao and S. Mao, "Research on ill-conditioned problem and modeling precision in GM(1,1) model," in *Proc. IEEE Int. Conf. Syst. Man Cybern.*, vol. 1, pp. 622–627, 2005, doi: 10.1109/ICSMC.2005.1571216.

[19] A. Stanoyevitch, *Introduction to Numerical Ordinary and Partial Differential Equations Using MATLAB*. Hoboken, NJ, USA: Wiley, 2011.

[20] D. S. Parker, P. R. Eggert, and B. Pierce, "Monte Carlo arithmetic: A framework for the statistical analysis of roundoff error," Tech. Rep. CSD-970014, University of California (Los Angeles, CA US), Los Angeles, 1997, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.6485

[21] E. Goubault and S. Putot, "Static analysis of numerical algorithms," in *Proc. 13th Int. Static Anal. Symp.*, 2006, pp. 18–34.

[22] L. Jiang and Z. Su, "Osprey: A practical type system for validating dimensional unit correctness of C programs," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 262–271, doi: http://doi.acm.org/10.1145/1134285.1134323.

[23] F. Chatelin, "Sur le taux de fiabilité général de la méthode CESTAC," *Comptes Rendus de l Académie des Sciences-Series I*, vol. 16, pp. 851–854, Jan. 1988.

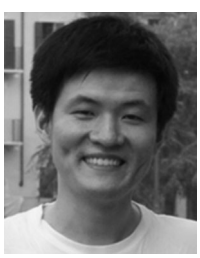
[24] P. R. Eggert and D. S. Parker, "Perturbing and evaluating numerical programs without recompilation: The wonglediff way," *Softw. Practice Experience*, vol. 35, no. 4, pp. 313–322, 2005, doi: http://dx.doi.org/10.1002/spe.637.

[25] D. S. Parker, "Monte Carlo arithmetic: Exploiting randomness in floating-point arithmetic," UCLA Computer Science Department, May 1997. [Online]. Available: http://web.cs.ucla.edu/~stott/mca/

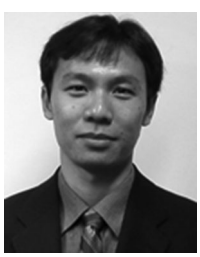
[26] M. Galassi, et al., *Gnu Scientific Library Reference Manual*, 1.2 ed., Godalming, U.K.: Network Theory Ltd., 2002.

[27] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei, "A genetic algorithm for detecting significant floating-point inaccuracies," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 20–22.

[28] N. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., Philadelphia, PA, USA: Soc. Ind. Appl. Math., 2002.
[29] W. Miller, "Toward mechanical verification of properties of roundoff error propagation," in *Proc. ACM Symp. Theory Comput.*, 1973, pp. 50–58.
[30] W. Miller, "Software for roundoff analysis," *ACM Trans. Math. Softw.*, vol. 1, no, 2, pp. 108–128, 1975.
[31] W. Miller and D. Spooner, "Software for roundoff analysis, II," *ACM Trans. Math. Softw.* vol. 4, no. 4, pp. 369–387, 1978, doi: http://doi.acm.org/10.1145/356502.356496.
[32] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. New York, NY, USA: Dover, 1994.
[33] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*, 1 ed. Philadelphia, PA, USA: Soc. Ind. Appl. Math., 2009.
[34] J. Stolfi and L. H. Figueiredo, "An introduction to affine arithmetic," *TEMA Tend. Mat. Apl. Comput.*, vol. 4, no. 3, pp. 297–312, 2003.
[35] S. Putot, E. Goubault, and M. Martel, "Static analysis-based validation of floating-point computations," *Numerical Software with Result Verification*, R. Alt, A. Frommer, R. B. Kearfott, W. Luther (eds), Berlin, Germany: Springer, 2004, pp. 306–313. [Online]. Available: http://link.springer.com/chapter/10.1007/978–3-540-24738-8_18
[36] E. Goubault, "Static analyses of the precision of floating-point operations," in *Proc. 8th Int. Static Anal. Symp.*, 2001, pp. 234–259.
[37] F. Benz, A. Hildebrandt, and S. Hack, "A dynamic program analysis to find floating-point accuracy problems," *SIGPLAN Not.*, vol. 47, no. 6, pp. 453–462, Jun. 2012. doi: 10.1145/2345156.2254118.
[38] T. Bao and X. Zhang, "On-the-fly detection of instability problems in floating-point program execution," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Languages & Appl.*, 2013, pp. 817–832, doi: 10.1145/2509136.2509526. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509526
[39] W. Miller and D. Spooner, "Automatic generation of floating-point test data," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 3, pp. 223–226, Sep. 1976, doi: 10.1109/TSE.1976.233818.
[40] R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb, "Symbolic path-oriented test data generation for floating-point programs," in *Proc. IEEE 6th Int. Conf. Softw. Testing Verification Validation*, 2013, pp. 1–10, doi: 10.1109/ICST.2013.17.
[41] S. Boldo and J. C. Filliatre, "Formal verification of floating-point programs," in *Proc. 18th IEEE Symp. Comput. Arithmetic*, 2007, pp. 187–194, doi: 10.1109/ARITH.2007.20.
[42] S. Boldo and G. Melquiond, "Flocq: A unified library for proving floating-point algorithms in CoQ," in *Proc. 20th IEEE Symp. Comput. Arithmetic*, 2011, pp. 243–252, doi: 10.1109/ARITH.2011.40.
[43] A. Ayad and C. Marché, "Multi-prover verification of floating-point programs," in *Automated Reasoning*, J. Giesl, R. Hhnle (eds.) Berlin, Germany: Springer, 2010, pp. 127–141. [Online]. Available: http://link.springer.com/chapter/10.1007/978–3-642-14203-1_11
[44] E. Darulova and V. Kuncak, "Trustworthy numerical computation in scala," *Proc. ACM Int. Conf. Object Oriented Program. Syst. Languages Appl.*, 2011, pp. 325–344. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048094
[45] M. Martel, "Propagation of roundoff errors in finite precision computations: A semantics approach," in *Proc. 11th Eur. Symp. Program.*, 2002, pp. 194–208.
[46] D. Monniaux, "The pitfalls of verifying floating-point computations," *ACM Trans. Program. Lang. Syst.* vol. 30, no. 3, pp. 12:11–12:41, May 2008, doi: 10.1145/1353445.1353446.
[47] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically adapting programs for mixed-precision floating-point computation," *Proc. 27th Int. ACM Conf. Int. Conf. Supercomputing*, 2013, pp. 369–378, doi: 10.1145/2464996.2465018. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465018
[48] C. Rubio-Gonzlez,, et al., "Precimonious: Tuning assistant for floating-point precision," *Proc. Int. Conf. High Performance Comput., Netw., Storage Anal.*, 2013, pp. 27:1–27:12, doi: 10.1145/2503210.2503296. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503296
[49] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," *Proc. 35th ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2014, pp. 53–64, doi: 10.1145/2594291.2594302. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594302
[50] K. Briggs, "Implementing exact real arithmetic in Python, C++ and C," *Theoretical Comput. Sci.* vol. 351, no. 1, pp. 74–81, Feb. 2006, doi: 10.1016/j.tcs.2005.09.058.
[51] J. Marcial-Romero andM. Escardo, "Semantics of a sequential language for exact real-number computation," *Proc. 19th Annu. IEEE Symp. Logic Comput. Sci.*, 2004, pp. 426–435, doi: 10.1109/LICS.2004.1319637.
[52] V. Ménissier-Morain, "Arbitrary precision real arithmetic: Design and algorithms," *J. Logic Algebraic Program.* vol. 64, no. 1, pp. 13–39, Jul. 2005, doi: 10.1016/j.jlap.2004.07.003.
[53] J. Blanck, "Exact real arithmetic using centred intervals and bounded error terms," *J. Logic Algebraic Program.* vol. 66, no. 1, pp. 50–67, Jan. 2006, doi: 10.1016/j.jlap.2005.07.002.
[54] N. Revol and F. Rouillier, "Motivations for an arbitrary precision interval arithmetic and the MPFI library," *Reliable Comput.* vol. 11, no. 4, pp. 275–290, Aug. 2005, doi: 10.1007/s11155–005-6891-y.
[55] M. Anderson, S. Tsen, L. K. Wang, K. Compton, and M. Schulte, "Performance analysis of decimal floating-point libraries and its impact on decimal hardware and software solutions," in *Proc. IEEE Int. Conf. Comput. Des.*, 2009, pp. 465–471, doi: 10.1109/ICCD.2009.5413114.
[56] X. Wang, S. Braganza, and M. Leeser, "Advanced components in the variable precision floating-point library," in *Proc. 14th Annu. IEEE Symp. Field-Programmable Custom Comput. Mach.*, 2006, pp. 249–258, doi: 10.1109/FCCM.2006.21.
[57] X. Wang and M. Leeser, "VFloat: A variable precision fixed- and floating-point library for reconfigurable hardware," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 3, pp. 16:1–16:34, Sep. 2010, doi: 10.1145/1839480.1839486.
[58] E. El-Araby, I. Gonzalez, and T. El-Ghazawi, "Bringing high-performance reconfigurable computing to exact computations," in *Proc. Int. Conf. Field Programmable Logic Appl.*, 2007, pp. 79–85, doi: 10.1109/FPL.2007.4380629.
[59] T. Granlund, "The GMP development team," *Gnu MP GNU Multiple Precision Arithmetic Library*, 2014. [Online]. Available: http://gmplib.org/
[60] C. Kormanyos, "Algorithm 910: A portable c++ multiple-precision system for special-function calculations," *ACM Trans. Math. Softw.* vol. 37, no. 4, pp. 45:1–45:27, Feb. 2011, doi: 10.1145/1916461.1916469.
[61] V. Shoup, NTL: A Library for doing Number Theory, 2014. [Online]. Available: http://www.shoup.net/ntl/
[62] B. Lambov, "RealLib: An efficient implementation of exact real arithmetic," *Math. Struct. Comput. Sci.* vol. 17, no. 1, pp. 81–98, Feb. 2007, doi: 10.1017/S0960129506005822.
[63] J. Vignes and M. La Porte, "Error analysis in computing," in *Proc. Int. Federation Inf. Process. Conf.*, 1974, pp. 610–614.
[64] D. Parker, B. Pierce, and P. Eggert, "Monte Carlo arithmetic: How to gamble with floating point and win," *Comput. Sci. Eng.*, vol. 2, no. 4, pp. 58–68, 2000, doi: 10.1109/5992.852391.

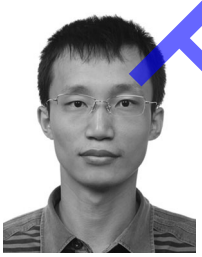


Enyi Tang received the BS and PhD degrees in computer science from Nanjing University, Jiangsu, China, in 2005 and 2013, respectively. Between 2008 and 2009, he visited as a research student with the University of California, Davis. He is now an associate professor in the Software Institute, Nanjing University. His main research interests are program analysis and software testing, in particular developing analysis and testing tools to improve software reliability and programming productivity.



Xiangyu Zhang received the PhD degree from the University of Arizona, in 2006. He is a professor with Purdue University. He works on dynamic and static program analysis and their applications in debugging, testing, forensic analysis, and data processing. He is currently a Purdue University scholar. He has received the 2006 ACM SIGPLAN Distinguished Doctoral Dissertation Award, NSF Career Award, ACM SIGSOFT Distinguished Paper Awards, Best Student Paper Award on USENIX'14, Best Paper Award on CCS'15 and Distinguished Paper Award on NDSS'16.

Norbert Th. Müller received the PhD degree from the Department of Computer Science, FernUniversität Hagen, Germany, in 1988. He is currently an extraordinary professor in the Department of Computer Science, Universität Trier, Germany. His primary research interests are computability and complexity on the field of real numbers, with an emphasis on efficient implementations.



Zhenyu Chen received the bachelor's and PhD degrees in mathematics from Nanjing University. He is currently a full professor in Software Institute, Nanjing University. His research interests focus on software analysis and testing. He has more than 80 publications at major venues including the *ACM Transactions on Software Engineering and Methodology*, the *IEEE Transactions on Software Engineering*, ICSE, FSE, ISSTA, ASE, ICST, etc. He has served as a PC co-chair of QRS 2016, QSIC 2013, AST 2013, IWPD 2012, and a program committee member of many international conferences. He has more than 30 patents and some of them have been used in Baidu, Alibaba, Huawei and so on. He is the founder of mooctest.net.

Xuandong Li received the BS, MS, and PhD degrees in computer science from Nanjing University in 1985, 1991 and 1994, respectively. Since 1994 he has been in the Department of Computer Science and Technology, Nanjing University where he is currently a professor. His research interests include formal support for design and analysis of reactive, disturbed, real-time, and hybrid systems, software testing and verification, and model driven software development.