Perturbing Numerical Calculations for Statistical Analysis of Floating-Point Program (In)Stability

Enyi Tang† Earl Barr‡ Xuandong Li† Zhendong Su‡

†State Key Laboratory for Novel Software Technology
Nanjing University, Nanjing, China
{eytang, lxd}@seg.nju.edu.cn

‡Department of Computer Science
University of California, Davis, USA
{etbarr,su}@ucdavis.edu

ABSTRACT

Writing reliable software is difficult. It becomes even more difficult when writing scientific software involving floating-point numbers. Computers provide numbers with limited precision; when confronted with a real whose precision exceeds that limit, they introduce approximation and error. Numerical analysts have developed sophisticated mathematical techniques for performing error and stability analysis of *numerical algorithms*. However, these are generally not accessible to application programmers or scientists who often do not have in-depth training in numerical analysis and who thus need more automated techniques to analyze their code.

In this paper, we develop a novel, practical technique to help application programmers (or even numerical experts) obtain high-level information regarding the numerical stability and accuracy of their code. Our main insight is that by systematically altering (or *perturbing*) the underlying numerical calculation, we can uncover potential pitfalls in the numerical code. We propose two complementary perturbations to statistically measure numerical stability: *value perturbation* and *expression perturbation*. Value perturbation dynamically replaces the least significant bits of each floating-point value, including intermediate values, with random bits to statistically induce numerical error in the code. Expression perturbation statically changes the numerical expressions in the user program to mathematically equivalent (in the reals, likely not in floating-point numbers), but syntactically different forms. We then compare the executions of these "equivalent" forms to help discover and remedy potential instabilities. Value perturbation can overstate error, while expression perturbation is relatively conservative, so we use value perturbation to generate candidates for expression perturbation. We have implemented our technique, and evaluation results on various programs from the literature and the GNU Scientific Library (GSL) show that our technique is effective and offers a practical alternative for understanding numerical stability in scientific software.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Statistical methods*; D.2.5 [**Software Engineering**]: Testing and Debugging; G.1.0 [**Numerical Analysis**]: General—*stability*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSTA'10 July 12–16, 2010, Trento, Italy

Copyright 2010 ACM 978-1-60558-719-823-0/10/07 ...$10.00.

General Terms

Languages, Reliability

Keywords

Floating-point, Numerical Code, Stability, Perturbation, Testing

1. INTRODUCTION

Because of Moore's law, the modern desktop PC has more computing power than a ten-year old supercomputer. Numerical models that were previously out of reach are now being used in the physics engines of games and in business analysis. As a result, a new class of programmers is writing numerical programs for a new class of users. These programmers will want to write custom numeric code not covered by existing numeric libraries. Programmers new to numerical computing tend to suffer from misconceptions that arise from thinking in terms of $\mathbb{R}$, not its floating-point approximation on computers. Even when these programmers have the requisite training, they will rarely have the time to evaluate the stability of their numerical code. At the same time, their users will be less able to diagnose and report errors when they occur. Our goal is to help these programmers (and their users).

Scientific programs are difficult to get right [9, 12]. Although numerical analysts have extensively studied numerical stability of algorithms, few practical tools exist that can automatically analyze an implementation to measure its stability. In this paper, we propose a novel, practical framework to help application programmers gain high-level knowledge about their numerical implementations. The goal is to warn programmers of unsuspected pitfalls in the numerical aspect of their code, such as unstable and inaccurate calculations. Scientists increasingly rely on numeric code to conduct their experiments; tools, such as ours, could improve their productivity.

1.1 Perturbation

Our framework rests on the following observation: it is possible to gain valuable information regarding a numerical calculation by systematically *perturbing* how the calculation is performed. We combine two complementary perturbations in our framework: *value perturbation* and *expression perturbation*. Value perturbation offers a *dynamic* view as it alters the computed values during program execution, while expression perturbation offers a *static* perspective as it alters the numerical expressions in a program. The basic concept of program transformation has been explored in prior research [5, 18, 19, 25] for evaluating accuracy of floating-point computations. However, the proposed approaches are less general, and few has been empirically demonstrated to be effective. We show for the first time how the unified value-level, dynamic perturbation, and expression-level, static perturbation, can aid programmers in understanding their numerical calculations.

Value perturbation operates on the floating-point representation of a value. The goal is to uncover, make manifest, the intrinsic errors imposed by floating-point approximations: representational and computational error. An example of computational error occurs when floating-point precisely represents a and b, but not ab. To uncover intrinsic error, we randomly perturb the low order bits of values, such as a and b. We do so by statically rewriting all reads and operations on floating-point values to replace a suffix of their significand with a random bit string. After perturbation, each run of the program generates different results. After multiple runs, we statistically analyze the results to determine the stability of the program. Our value perturbation is similar to Monte Carlo Arithmetic (MCA) [25]. We extend that framework with expression perturbation. We also provide a practical realization and an empirical evaluation to show its practical benefits.

Expression perturbation operates on the syntax of an expression. Applications of the associative, distributive, and commutative laws can generate a large number of syntactically different expressions. Each expression in this space of variants is equivalent over the reals. It is well-known that over floating-point arithmetic, however, not all of these variants are equivalent. Each variant induces a direct implementation in floating-point arithmetic. For a well-conditioned problem, some of the direct implementations are *stable*, *viz.* small changes in their input cause small changes in their output; the rest are *unstable*. We explore the variant space of well-conditioned expressions to check numerical implementations for potential instability. Martel also explored this variant space and proposed an abstract interpretation-based technique [18, 19] to extract the most stable, equivalent variant. However, this technique, due to its very nature, suffers from scalability and precision issues, and has not been shown practical. Our approach is statistical, and is therefore simpler and more practical for analyzing stability (but it does not discover stable variants). We systematically generate an expression's equivalent forms over the reals and evaluate these variants to gain statistical information about a calculation. To perturb a program, we statically perturb its constituent expressions in sequence.

1.2 Usage

In our framework, a user selects expression or value perturbation and compiles their program. The resulting binary incorporates the selected perturbation(s). When executed, the perturbed program runs the original program to completion multiple times; the user specifies that number of runs. We report two summary statistics across the results of those runs: the maximal difference (MD) and the coefficient of variation (CV). When identifying unstable algorithms, it is prone to false positives, as Kahan has observed [14]. Our results show that expression perturbation is much less prone to false positives. Value perturbation is faster to evaluate because, unlike expression perturbation, it creates a single variant. Thus, we apply our perturbations in two stages: first we employ value perturbation to generate candidates whose expressions we then perturb.

In the physical sciences, publications rarely report experimental results that were directly observed in the laboratory. Instead, scientists directly measure one quantity and use the outcome to calculate what the value of another, more interesting, physical quantity must be. The measurement apparatus and the number of experimental trials engender errors in the directly observed quantity. A direct lab result has the form $x_0 \pm \Delta x$. If the quantity to be published is $z = f(x)$, the experimenter will determine $\frac{df}{dx}$ and calculate the error on the reported quantity Δz, as $\Delta x \frac{df}{dx}|_{x=x_0}$. When a numeric program is used, that program is a source of systematic error due to finite precision and statistical error, if it uses randomness. Thus, error propagation dictates that a scientist must account for this error.

```
1  double naive(double[] samples, int n) {
2     double sum, squaresum;
3     for (int i; i < n; i++) {
4        sum += samples[i];
5        squaresum += samples[i] * samples[i];
6     }
7     return (squaresum - sum * sum / n) / n;
8  }
```

Figure 1: Naïve variance algorithm.

A recent paper in the Physical Review D reported $-37 \pm 88^{\text{stat}} \pm 14^{\text{syst}}$ MeV$/c$ [1], where 88^{stat} denotes statistical error, 14^{syst} denotes systematic error, and MeV$/c$ is megaelectron volts per c, the speed of light in a vacuum. Assume the authors used numeric code in their experimental procedure. Let α be the statistical error the authors assigned to that numeric code and β be the systematic error; then α contributes to the derivation of 88^{stat} MeV$/c$ and β to 14^{syst} MeV$/c$, the final, reported errors. The authors could have used our tool to find coefficient of variation (CV) and z, the central value of sample set produced by multiple runs of their numeric code under perturbation, and checked their error calculation with

$$|CV| \leq \left| \frac{\sqrt{\alpha^2 + \beta^2}}{z} \right|.$$

Adding errors in quadrature as shown here is standard practice in error analysis [7, Equation 3.73].

1.3 Contributions

We have implemented our perturbation framework and evaluated it on various programs from the literature and the GNU Scientific Library (GSL). Results show that our technique is effective and able to identify instabilities in the test programs and validate the stability of others. We believe it offers a practical alternative for understanding numerical stability of scientific software.

This paper makes the following main contributions:

- We propose a novel framework to statistically analyze the numerical stability of scientific code. The framework is based on the general notion of perturbation and consists of two complementary techniques: expression and value perturbations.
- We present detailed algorithms and optimizations on how to realize expression and value perturbations.
- We have implemented our technique and empirically evaluated it on various test subjects to show its effectiveness.

The rest of the paper is structured as follows. We show how our technique works on a simple, but real, example in Section 2. In Section 3, we formalize our technique and present our perturbation algorithms. After describing the implementation in Section 4, we show evaluation results in Section 5 and discuss related work in Section 6. Finally, we conclude and discuss future work (Section 7).

2. EXAMPLE

This section shows how value and expression perturbations can help programmers uncover and diagnose numerical instabilities.

A fundamental problem in numerical computation is the efficient and accurate calculation of statistical variance. Figure 1 shows a naïve, one-pass variance implementation. For each sample, it updates the running sums of both the samples and their squares. When we evaluate this code with 1000 inputs of 3.1415926×10^{10}, the output should be 0. However, compiled using `gcc 4.2.4` and run under Ubuntu 8.04, the program outputs -1.27446×10^7.

```
  1   squaresum * (1/n) + (-sum) * sum
                                 * (1/n) * (1/n)
  2   squaresum * (1/n) + sum * (1/n)
                                 * (1/n) * (-sum)
  3   (1/n) * (sum * (1/n) * (-sum) + squaresum)

                         ...
143   (1/n) * sum * (1/n) * (-sum)
                                 + squaresum * (1/n)
144   (1/n) * sum * (1/n) * (-sum)
                                 + (1/n) * squaresum
```

Figure 2: Selected perturbed expressions.

```
 1   void knuth(double[] samples, int n) {
 2      double delta, mean, M2;
 3      mean = M2 = 0;
 4      for (int i = 0; i < n; ++i) {
 5         delta = samples[i] - mean;
 6         mean += delta / i;
 7         M2 += delta * (samples[i] - mean);
 8      }
 9      return M2 / n;
10   }
```

Figure 3: Knuth's stable variance algorithm.

This result is obviously wrong and the code is unstable, indeed notoriously so — this one-pass algorithm is a staple example of an unstable algorithm in numerical analysis texts. The problem is that `squaresum` and `sum * sum / n` are very close. How does a programmer use our technique to automatically discover unstable programs like this one?

Value perturbation runs quickly, because it creates a single variant of the tested program, and tends to exaggerate error: if a floating-point function appears stable under value perturbation, it is likely to be stable. Thus, the programmer first evaluates Figure 1 under value perturbation, which reports 2.228E+8 MD and 2.060 CV when only one bit is perturbed.

As expected, this result suggests that Figure 1 is unstable. To confirm this suspicion, the programmer turns to expression perturbation, which is a more expensive and more conservative test, one less prone to false positives. Expression perturbation transforms an expression to one of many possible forms equivalent over $\mathbb{R}$. In particular, the expression at line 7 of Figure 1 has 144 equivalent expressions, some of which are depicted in Figure 2. Each of the 144 variants induces a variant of the original function. We select a random subset of these function variants and evaluate them on the original input of 1000 samples of 3.1415926×10^{10}. The CV of line 7 is 7.96E-39. Since this CV is quite small, syntactically rewriting this expression to alter its order of evaluation is unlikely to be fruitful.

Taken together, these perturbation results suggest that either a higher precision floating-point format is needed or the programmer may even need to entirely rewrite the code using a more stable algorithm. Here, we know that the naïve implementation is the culprit, and a new algorithm is needed. Fortunately, Knuth presented just such a stable algorithm for variance calculation [15], shown in Figure 3. Given the same inputs and run on the same platform, this algorithm returns *exactly* 0. Under value perturbation, its MD is 3.715E+3 and its CV is 0.542 and its accuracy is robust in the face of expression perturbation. Both of these results are clearly better than those reported for Figure 1.

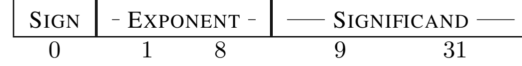
3. PERTURBATION ALGORITHMS

Our value and expression perturbations rewrite a program that uses floating point to elicit any latent propensity to stability or propagation errors the program may possess. At the high-level, we rewrite the program to make stability or propagation errors more likely to occur if the program is susceptible to them. The goal of our perturbations is to uncover oversights or misconceptions in the use of floating-point arithmetic and to measure how robust an implementation is in the face of numerical errors. When the result of a floating-point computation is stable in the face of our perturbations, the computation is likely to be free of stability errors. For value perturbation, we describe how to introduce controlled, random errors inside numerical calculations. For expression perturbation, we show how to systematically generate mathematically equivalent expressions of a given numerical expression e. Value perturbation offers a *dynamic* perspective as it focuses on the underlying computed values, while expression perturbation offers an alternative and complementary, *static* perspective that focuses on the expression level. In this section, we define our perturbations and present algorithms that realize them.

$$e \ ::= \ -e \ | \ e+e \ | \ e-e \ | \ e \times e \ | \ e \,/\, e \ | \ \mathcal{E} \ | \ v$$

Figure 4: Grammar G for floating-point expressions.

Single Precision



Double Precision

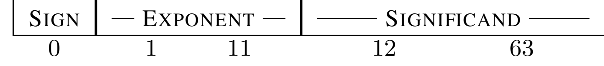


Figure 5: Two IEEE 754 floating-point representations.

We assume that the numerical programs we perturb have the form $P : I \to \mathbb{R}$, where I denotes the input domain. To handle programs of the form $P : I \to \mathbb{R}^n$, we decompose them into a set of n functions whose form is $P_i : I \to \mathbb{R}$. Restricting our attention only to its floating-point computations, the program P computes $r = e_0 \odot e_1 \odot \cdots \odot e_{m-1}$, where $\odot$ denotes an arbitrary floating-point operation.

Since we are only interested in floating-point expressions, we do not rewrite any component of the program we are perturbing other than a floating-point value or expression. Floating-point expressions may contain other programming language elements ($\mathcal{E}$), such as a function call. The fact that an $\mathcal{E}$ appears in a floating-point expression implies that it evaluates to a floating-point expression. Figure 4 depicts G, the grammar of floating-point expressions that we rewrite. The v in G denotes a primitive floating-point value.

3.1 Value Perturbation

Floating-point computations often have roundoff errors. Our value perturbation induces and exaggerates roundoff error: if the computation survives this perturbation, it is likely to survive any transient error it faces in practice.

When s is the sign; b is the base with the value 2; p is the number of digits in the significand; x is the exponent; and f is the integer value of the numerator of the significand, we have

$$(-1)^s \times \frac{f}{b^{p-1}} \times b^x \qquad (1)$$

Figure 5 depicts two formats prescribed by the IEEE 754 Standard for Floating-Point Arithmetic [2].

Each $r \in \mathbb{R}$ has a corresponding floating-point representation f, which is subject to representational error. To approximate the error in f, we replace a k-width suffix of f's significand with a random k-bit string; in effect, we represent r in floating-point as the prefix of f's significand concatenated with any of its 2^k possible suffixes

```
1   double p_v (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   }
```

Figure 6: Implementation of p_v.

and force the representational error to be uniformly distributed. Since we perturb a k-width suffix, our technique is more efficient than calculating $r(1 \pm \kappa)$ because our approach requires fewer operations and focuses on the significand. When F denotes all possible floating-point values in a particular floating-point format, let $p_v : \mathbb{N}_1 \times F \rightarrow F$, in Figure 6, denote this perturbation.

The expression transformation function

$$p(e) = \begin{cases} p_v(v), & e = v \\ p_v(eval(\mathcal{E})), & e = \mathcal{E} \\ -p(e_1), & e = -e_1 \\ p_v(p(e_1) + p(e_2)), & e = e_1 + e_2 \\ p_v(p(e_1) - p(e_2)), & e = e_1 - e_2 \\ p_v(p(e_1) \times p(e_2)), & e = e_1 \times e_2 \\ p_v(p(e_1) \,/\, p(e_2)), & e = e_1 \,/\, e_2 \end{cases} \tag{2}$$

recursively applies p_v to each primitive floating-point value and the result of each operation. Function $eval$ evaluates an $\mathcal{E}$ and returns the floating-point expression that $\mathcal{E}$ denotes. For clarity, we have implicitly curried both p and p_v to elide n, the length of the permuted suffix.

Randomly perturbing a suffix of floating-point values allows us to statistically model errors introduced by floating-point approximations. We not only model representational errors, but also computational errors since we also perturb intermediate results.

When we apply the transformation function p to the expression on Line 7 in Figure 1, it returns an expression that contains the factor $p_v(p_v(\texttt{sum}) \times p_v(\texttt{sum})/p_v(\texttt{n})))$. Capturing each perturbation as ϵ_i and letting $\epsilon_5 = \epsilon_0\epsilon_3 + \epsilon_1\epsilon_3 + \epsilon_0\epsilon_1\epsilon_3$, algebraically this factor is

$$\frac{\texttt{sum}^2(1+\epsilon_5)}{\texttt{n}(1+\epsilon_2)}(1+\epsilon_4). \tag{3}$$

3.2 Expression Perturbation

We perturb expressions to forms that are equivalent over $\mathbb{R}$, but syntactically different, to see how they affect the stability of floating-point computations. In this section, we first present our algorithms to generate the equivalent expressions for a given expression e. The number of expressions these algorithms generate may be very large, so we also present a Monte Carlo method for an unbiased sampling of these expressions.

3.2.1 *Exhaustive Expression Generation*

We first define a canonical form for expressions that our algorithms assume. This form is completely unfolded, *i.e.* it cannot be rewritten by the distributive law. Then we present algorithms for parenthesizing and applying the associative and commutative laws to an expression. Our factoring algorithm folds, or factors, a distributed expression. Finally, we enumerate all expressions by feeding the result of the association and commutation algorithm to our factoring algorithm.

(R1)	$\Theta(v) \rightarrow v$
(R2)	$\Theta(\mathcal{E}) \rightarrow \mathcal{E}$
(R3)	$\Theta(-(-e_1)) \rightarrow \Theta(e_1)$
(R4)	$\Theta(-(e_1+e_2)) \rightarrow \Theta(-e_1-e_2)$
(R5)	$\Theta(-(e_1 \times e_2)) \rightarrow \Theta(-e_1 \times e_2)$
(R6)	$\Theta(-e_1) \rightarrow -\Theta(e_1)$
(R7)	$\Theta(e_1+e_2) \rightarrow \Theta(e_1)+\Theta(e_2)$
(R8)	$\Theta(e_1-e_2) \rightarrow \Theta(e_1)+(\Theta(-e_2))$
(R9)	$\Theta(e_1 \times (e_2 \pm e_3)) \rightarrow \Theta(e_1 \times e_2)+\Theta(\pm e_1 \times e_3)$
(R10)	$\Theta((e_1 \pm e_2) \times e_3) \rightarrow \Theta(e_1 \times e_3)+\Theta(\pm e_2 \times e_3)$
(R11)	$\Theta(e_1 \times e_2) \rightarrow \Theta(e_1) \times \Theta(e_2)$
(R12)	$\Theta(e_1 \,/\, e_2) \rightarrow \Theta(e_1) \times (1 \,/\, \Theta(e_2))$

Figure 7: Canonicalizing term rewriting system Θ.

DEFINITION 3.1 (CANONICAL FORM). *An expression e over the grammar G (Figure 4),* viz. *$e \in L(G)$, is* canonical *when:*

(P1) It is minimal*: with no additive zeros or multiplicative ones;*
(P2) It contains no binary subtraction operators;
(P3) It uses division only to form inverses; and
(P4) It cannot be rewritten by the distributive law.

Given an expression e that is minimal with respect to additive zeros and multiplicative ones, we rewrite e to its canonical form. Figure 7 defines a term rewriting system Θ that rewrites a minimal expression to its canonical form. Θ uses the equivalences $X - Y = X + (-Y)$ and $X/Y = X \times (1/Y)$ to realize properties P2 and P3. To realize P4, we apply Θ recursively, thereby successively applying the distributive law to expand all the polynomials, so that we need only consider factoring below.

Our canonical form eliminates binary subtraction and restricts division to inverse, thus we consider only addition and multiplication, *i.e.* the set of operators $\{+, \times\}$, in the ensuing discussion.

To illustrate the operation of Θ, we apply it to

$$e = (x+y) \times (a-b). \tag{4}$$

Θ applies each rule is succession, until it reaches R9, which unifies with e, producing $\Theta((x+y) \times a) + \Theta(-(x+y) \times b)$. R10 then separately unifies with each of these Θ applications. The recursion terminates with

$$\Theta(e) = x \times a + y \times a + (-x \times b + (-y \times b)). \tag{5}$$

An expression forms a tree whose interior nodes are operators and whose leaves are either floating-point values or uninterpreted grammar elements, $\mathcal{E}$.

DEFINITION 3.2 (EXPRESSION UNIT). *An* expression unit *is a subtree whose root operator label does not match the operator label of the root of the entire expression tree.*

The function $units : E \rightarrow 2^E$ returns the set of expression units formed from an expression. The function $root : E \rightarrow \{+, \times\}$ returns the operator at the root of its input's expression tree.

Algorithm 1 defines the function $z : \mathcal{S} \times \{+, \times\} \rightarrow 2^E$. Given a permutation of expression units, z recursively bifurcates that permutation to enumerate all ways to parenthesize the permutation relative to the given operator.

Algorithm 2 defines $t_{ac} : E \rightarrow 2^E$. This function uses the associative and commutative laws to generate all the mathematically equivalent forms of its input over the reals. The base case of t_{ac}

Algorithm 1 $z : \mathcal{S} \times \{+, \times\} \rightarrow 2^E$ (All Parenthesizations)

input $s \in \mathcal{S}, \odot \in \{+, \times\}$
1: $T \leftarrow \emptyset$
2: **if** $s \neq \epsilon$ **then**
3: $\forall s_1, s_2 . s_1 s_2 = s$ **do**
4: $T \leftarrow \{(t_1 \odot t_2) \mid t_1 \in z(s_1, \odot), t_2 \in z(s_2, \odot)\} \cup T$
5: **end for**
6: **end if**
7: **return** T

Algorithm 2 $t_{ac} : E \rightarrow 2^E$
(Associative and Commutative Transformation)

input $e \in E$
1: **if** $e = v \vee e = \mathcal{E}$ **then**
2: **return** $\{e\}$
3: **end if**
4: $U \leftarrow units(e)$ // The set of all expression units of e.
5: $T \leftarrow \emptyset$
6: $\forall s \in$ all permutations of U **do**
7: $T \leftarrow T \cup z(s, root(e))$
8: **end for**
9: $\forall u_i \in U$ **do**
10: $S \leftarrow t_{ac}(u_i)$
11: $T \leftarrow \bigcup_{u_i' \in S} T[u_i \rightarrow u_i']$
12: **end for**
13: **return** T

returns primitives as a set. Otherwise, it extracts the expression's units at line 4. In Equation 5, $U = \{x \times a, y \times a, -x \times b, (-y \times b)\}$. At line 7, t_{ac} computes all parenthesizations of U. If any element of U is not primitive, we pick an arbitrary permutation of its units, then, line 10, generate all possible rewritings of those units that recursive calls to t_{ac} on each unit generate. Line 11, we return the set formed by replacing each unit with all its possible rewritings.

For $u_i \in units(e), n = |units(e)|$ and using Catalan function

$$C(x) = \frac{(2x)!}{x!(x+1)!} \tag{6}$$

which counts the parenthesizations that Algorithm 1 returns, the number of expressions t_{ac} generates is

$$N_{tac}(e) = \begin{cases} 1, & e \in \{v, \mathcal{E}\} \\ n! \times C(n-1) \times \prod_{i=1}^{n} N_{tac}(u_i), & \text{otherwise.} \end{cases} \tag{7}$$

This number grows very quickly; Algorithm 2 returns 1920 different expressions when applied to Equation 5.

Algorithm 3 defines $f : E \rightarrow 2^E$ which applies the distributive law to generate equivalent expressions over the reals. Applying f to each element of $t_{ac}(e)$ generates all expressions equivalent to e over the reals. The function t_{ac} preserves canonical form, so f only needs to factor the expressions t_{ac} outputs.

Definition 3.3 (Factor Point). *For $a, b, c, d \in L(G)$, the language of all valid expressions, a* factor point *is the $+$ operator in the expression $a \times b + c \times d$, where $a = c$ or $b = d$.*

In Algorithm 3, $fp : E \rightarrow 2^P$ at line 6 returns a set of all factor points in e'. Take the expression in Equation 5 as an example. The first and the third $+$ are both factor points but the second one is not. Each factor point can be factored. At line 8 we consider all

Algorithm 3 $f : E \rightarrow 2^E$ (Factoring)

input $e \in E$
precondition $e \neq v \wedge e \neq \mathcal{E}$
1: $E_l \leftarrow \{e\}, R \leftarrow \emptyset$
2: $P_o \leftarrow \emptyset$ // Old, already visited factor points.
3: **repeat**
4: $E_n \leftarrow \emptyset$
5: $\forall e' \in E_l$ **do**
6: $P_n \leftarrow fp(e') \setminus P_o$ // New factor points.
7: $\forall P_a \in 2^{P_n}$ **do**
8: $E_n \leftarrow E_n \cup \{factor(e', P_a)\}$
9: **end for**
10: $P_o \leftarrow P_o \cup P_n$
11: **end for**
12: $R \leftarrow R \cup E_l, E_l \leftarrow E_n$
13: **until** $E_l = \emptyset$
14: **return** R

Algorithm 4 Exhaustive Expression Generation

input $e \in E$
1: **if** $e = v \vee e = \mathcal{E}$ **then**
2: **return** $\{e\}$
3: **end if**
4: $R \leftarrow \emptyset, T \leftarrow t_{ac}(e)$
5: $\forall e' \in T$ **do**
6: $R \leftarrow R \cup f(e')$
7: **end for**
8: **return** R

possible factorings. New factor points arise during factoring. For example, if we apply factoring on both the first and the third $+$ operators of the expression in Equation 5, we get the expression $(x + y) \times a + (-(x + y) \times b)$. Then a new factor point, the second $+$, occurs. f iteratively finds new factor points in these expression until it cannot generate any new expressions.

Algorithm 4 applies Algorithm 3 and Algorithm 2 to generate all mathematically equivalent expressions for the expression e.

3.2.2 *Monte Carlo Expression Generation*

For an arbitrary expression, the space of syntactically different but minimal equivalent expressions over $\mathbb{R}$ is large. For $e = (x + y) \times (y + z) \times (z + x)$, $N_{tac}(e)$ is $7,437,513,790,586,880$ unique expressions. A practical, scalable technique cannot directly work with such large sets. Instead, we use the Monte Carlo method [6]: we work with subsets of these sets of expressions, chosen uniformly at random.

Definition 3.4 (Expression Number and Limit). *The* expression number *of e is $N_{tac}(e)$, the number of expressions equivalent to e over the reals using the commutative and associative laws; and the number of those expressions we wish to randomly sample is L,* expression limit *of e.*

Definition 3.5 (Expression Length). *The number of the leaves in an expression tree is the* length *of that expression. These leaves are either values or language grammar elements ($\mathcal{E}$).*

Three numbers decide N_{tac} in Algorithm 2: $N_p = k!$, the number of permutations of U at line 6;

$$N_z = C(k-1) = \frac{(2k-2)!}{(k-1)!k!}, \tag{8}$$

the number of parenthesizations at line 7; and N_r, the number of expressions recursively generated at line 10 and we have

$$N_{tac} = N_p \times N_z \times N_r. \tag{9}$$

We want to limit N_{tac} to L without introducing bias. That means we want the probability for each expression to be chosen is $\frac{L}{N_{tac}}$. We set L equal to three independent probability functions that determine whether or not to take a decision point. These functions,

$$L_p = L^{\frac{\ln N_p}{\ln N_{tac}}}, \quad L_z = L^{\frac{\ln N_z}{\ln N_{tac}}}, \quad L_r = L^{\frac{\ln N_r}{\ln N_{tac}}}, \tag{10}$$

distribute the probability uniformly across the locally available choices. The probability is $\frac{L_p}{N_p}$ for each permutation, $\frac{L_z}{N_z}$ for each parenthesization, and $\frac{L_r}{N_r}$ for each recursion. So the probability to select an arbitrary expression is

$$\frac{L_p}{N_p} \times \frac{L_z}{N_z} \times \frac{L_r}{N_r} = \frac{L}{N_{tac}} \tag{11}$$

We cannot directly compute N_r since the cost of the recursion is prohibitive for an arbitrary expression. The number of units across all recursive applications of t_{ac} determines N_r. The length of expression approximates the number of units. When n is the length of the input expression, k is the number of units of the input expression and $\mu = \ln N_{tac}$ (See Equation 15), we conclude

$$\frac{N_r}{N_p \times N_z} \approx (\frac{n}{k})^{\mu} \tag{12}$$

$$\begin{aligned} \frac{L_r}{L_p \times L_z} &= L^{\frac{\ln N_r - \ln N_p - \ln N_z}{\ln N}} = L^{\frac{\ln \frac{N_r}{N_p N_z}}{\mu}} \\ &\approx L^{\ln (\frac{n}{k})^{\mu}/\mu} = L^{\ln (\frac{n}{k})} \end{aligned} \tag{13}$$

We have $L = L_p \times L_z \times L_r$ from Equation 10. Since L_p and $L_r \times L_z$ are positive, with Equation 13 we have

$$\begin{aligned} L_r &= \sqrt{\frac{L_r}{L_p \times L_z} \times L} \approx \sqrt{L^{\ln (\frac{n}{k})} \times L} = L^{\ln \sqrt{\frac{ne}{k}}} \\ L_p \times L_z &= L/L_r \approx L^{\ln \sqrt{\frac{ke}{n}}} \end{aligned} \tag{14}$$

where e is Euler's number, not an expression.

Simultaneously solving Equation 10 and Equation 14, we derive

$$\mu = \ln N \approx \frac{\ln N_p N_z}{\ln \sqrt{\frac{ke}{n}}} \tag{15}$$

$$L_p \approx L^{\frac{\ln \sqrt{\frac{ke}{n}} \ln N_p}{\ln N_p N_z}} \tag{16}$$

$$L_z \approx L^{\frac{\ln \sqrt{\frac{ke}{n}} \ln N_z}{\ln N_p N_z}} \tag{17}$$

Although Equation 14 limits the total numbers of recursive calls at line 7 of Algorithm 2, we also need, for each recursion on the unit u_i, the limit

$$L_r = \prod_{i=1}^{k} L_{r,i}. \tag{18}$$

Again we use the length n_i of each unit u_i to estimate its limit, $L_{r,i}$. Thus,

$$n = \sum_{i=1}^{k} n_i. \tag{19}$$

Algorithm 5 $t'_{ac} : E \times \mathbb{N} \to 2^E$
(Optimized Associative and Commutative Transformation)

input $e \in E, L \in \mathbb{N}$
1: **if** $e \neq v \wedge e \neq \mathcal{E}$ **then**
2: **return** $\{e\}$
3: **end if**
4: $U \leftarrow units(e)$ // The set of all expression units of e.
5: $T \leftarrow \emptyset$
6: With $L, n = |e|, k = |units(e)|$,
solve Equation 16 to derive L_p and L_z.
7: $U_p \leftarrow$ all permutations of U
8: **for** $i := 0; i < L_p; i := i + 1$ **do**
9: randomly take $s \in U_p$
10: $U_p \leftarrow U_p - s$
11: $T \leftarrow T \cup z'(s, root(e), L_z)$
12: **end for**
13: **if** $\forall u \in U, u = v \vee u = \mathcal{E}$ **then**
14: **return** T
15: **else**
16: $\forall u_i \in U$ **do**
17: With $L, n_i = |u_i|$, solve Equation 20 to derive $L_{r,i}$.
18: $S \leftarrow t'_{ac}(u_i, L_{r,i})$
19: $T \leftarrow \bigcup_{u'_i \in S} T[u_i \to u'_i]$
20: **end for**
21: **return** T
22: **end if**

Algorithm 6 Value Perturbation Test Algorithm

input $f[e_1, \cdots, e_n]$ // The function-to-test
input I // A set of inputs to f
1: $\forall i \in I$ **do**
2: print $f[p(e_1)/e_1, \cdots, p(e_n)/e_n](p_v(i))$
3: **end for**

Then, we derive

$$L_{r,i} \approx L_r^{\frac{n_i}{n}} \approx L^{\frac{n_i}{n} \ln \sqrt{\frac{ne}{k}}}. \tag{20}$$

Algorithm 5, a sampling-optimized version of Algorithm 2, incorporates the limits defined in Equation 16 and Equation 20 to restrict its sampling of the space of possible expressions. At each limited decision point, its choice is nondeterministic. Thus, it effectively chooses a path in the recursion tree uniformly at random, and its results are therefore representative of the entire population. Algorithm 5, implements the function $t'_{ac} : E \times \mathbb{N} \to 2^E$. Its new parameter is the sampling limit. At line 6 we compute L_p, L_z according to Equation 16. And we use them to restrict the expressions generated at line 8 and line 11. The function z' in line 11 is the optimized version of Algorithm 1. We discuss it in Section 4. We compute Equation 20 to derive $L_{r,i}$ at line 17 and use the resulting values at line 18 to limit recursion.

4. IMPLEMENTATION

To implement our value perturbation, we needed only to inject calls to p_v, our value perturbation function, at the appropriate places in a program. The C transformer CIL [23] is particularly well-suited for this task. Algorithm 6 is the value perturbation we implemented and used in our evaluation. It applies p defined in Equation 2 in Section 3.1 to inject calls to p_v in every floating point expression in a program, then links p_v, and runs the program to generate the output. Figure 6 depicts the implementation of p_v in C.

Algorithm 7 Expression Perturbation Test Algorithm

input $f[e_1, \cdots, e_n]$ // The function-to-test
input I // A set of inputs to f
input $L \in \mathbb{N}$ // The limit defined in Definition 3.4
1: $\forall i \in I$ **do**
2: **for** $j \leftarrow 1; j \leq n; j \leftarrow j+1$ **do**
3: **for** $k \leftarrow 1; k \leq L; k \leftarrow k+1$ **do**
4: print $f[e_j/\text{perturb}(e_j)](i)$
5: **end for**
6: **end for**
7: **end for**

We implemented our expression perturbation on ROSE, a static analysis framework for C and C++ [26], because of its rich API for abstract syntax trees (AST). The bulk of the implementation uses the visitor design pattern to traverse an AST.

Let the *shape* of a parenthesization be the set of pairs of the indices of a matched set of parentheses. Our parenthesization algorithm z was used at line 7 in Algorithm 1 and its sampling-optimized variant z' was used at line 11 of Algorithm 5. Both feature an optimization: rather than directly parenthesizing their input sequence of units, they produce the set of all shapes for the length of their input sequence. Generating a set of shapes is much faster than rewriting the input sequence. Then they pick one shape from the set of shapes, uniformly at random subject to the limit L_z, and apply it to the input to produce a concrete parenthesization of the input.

Algorithm 7 is the expression perturbation algorithm we implemented and use in our evaluation. It operates on the lexicographic sequence of floating-point expressions that comprise the algorithm f. The "perturb" function is the composition of the algorithms Algorithm 2 and Algorithm 3; it takes a floating-point expression e and produces an expression e' that would be equivalent (*i.e.* $e = e'$) if e and e' were over the reals, not floating-point approximations of the reals. Conceptually, Algorithm 7 decomposes f into a set of n functions, $f_i : I \rightarrow \mathbb{R}$, for $f_i = e_1 \odot e_{i-1} \odot \text{perturb}(e_i) \odot e_{i+1} \odot \cdots \odot e_n$ where $1 \leq i \leq n$. Algorithm 7 perturbs only one expression at a time in sequence; it does not perturb arbitrary subsets of the set of expressions that f computes. Each f_i defines a set of functions, whose cardinality is L, the limit defined in Definition 3.4, where each function in the set has a different syntactic expansion of e_i. Regardless of the semantics of f, we know, for each input $i \in I$, that each function in f_i, over the reals, produces the same output, and therefore ideally should produce results that are closely clustered in floating-point. Thus, for each f_i and for each input $i \in I$, Algorithm 7 produces a set of results whose MD and CV we can then compute.

5. EMPIRICAL EVALUATION

Here we show that our value and expression perturbations find unstable expressions. Empirically, the maximal differences (MD) reported by our value perturbation are almost linear to the perturbation suffix length in logarithmic scale. Thus, we calculate and report the MD of only few suffix lengths. We also report the coefficient of variation (CV) for two reasons: 1) it is a normalized measure of variance that can be compared across distributions and, 2) in contrast to MD, it is an aggregate statistic that summarizes an entire sample set. CV shines in our value perturbation results, where it correctly identifies and partitions our test suite into stable and unstable programs. As Kahan noted, however, value perturbation can overstate error [14]; this problem manifests itself most spectacularly in Figure 13 where some functions exhibit maximal CV on the order of 120. As our data makes clear, expression perturbation is more conservative and can therefore check and confirm the classifications made by value perturbation.

We ran our evaluations on a workstation with 2 Intel Xeon CPU 3.00 GHz, and 2GB Physical Memory. The operating system is Ubuntu 7.10 and we used gcc 4.1.3.

5.1 Test Subjects from the Literature

We collected our first set of test subjects from related academic work. Figure 8 depicts the key segment of each of our tests. `Inv.c`, which computes the inverse of its input from Goubault [11], the `Newton.c` test, which computes $\sqrt{2}$ by Newton's method [5], and `sample_run.c`, a computation drawn from mechanical engineering simulator [13], are stable. Interestingly, `sample_run.c` is stable, under our testing, in spite of the large number of expressions, equivalent over $\mathbb{R}$, that it generates. In contrast, tests `exp.c`, which Martel used to test his analysis [18], `Poly.c`, a simple polynomial test, like `Inv.c` drawn from Goubault [11], and `root.c`, which solves a quadratic equation of one variable [25], are unstable. Test `exp.c` is unstable since it adds two operands of vastly different magnitude. A better way to write the problematic expression is `a*b+a*(c+d)`. `Poly.c` is unstable when `x` is close to 1 or 0 because of expression $z = (x-1)^4$. Test `root.c` is similar to Figure 1, the naive computation of variance in Section 2: it is unstable since `sqrt(b*b-4*a*c)` is near `b`, so subtracting them is error-prone. When the program itself does not have any floating-point output, we use the result of its last evaluated floating-point expression as the output.

5.1.1 Value Perturbation Results

Figures 9–11 depict the results of applying value perturbation to our test suite. In these figures, we vary the length of the perturbed suffix from one to sixteen bits. At zero, the induced perturbation is zero by definition, so we omit zero on the x-axis. At each length, we run the perturbed programs 1000 times. Figure 9 graphs the MD of value perturbation; Figure 11 and Figure 10 graph the CV.

For our test suite, Figure 9 demonstrates that the MD is almost linear in the length of the perturbed suffix when plotted in logarithmic scale. The CV clearly and *correctly* splits our test suite in two. In Figure 10, `Inv.c`, `Newton.c` and `sample_run.c` are stable; in Figure 11, `exp.c`, `Poly.c` and `root.c` are quite unstable. The MD with one bit perturbation of all programs in the unstable group is more than 3.00E-4, while for stable group it is less than $9.00E-12$. The CV for the unstable group is more than 8.00E-8. Indeed, when we perturb five bits, `root.c` produces 1.98E+1. In contrast, the CV of all programs in stable group is less than 9.00E-9. These results all match the properties of each program, depicted in Figure 8 and described above.

5.1.2 Expression Perturbation Results

Figure 12 presents the results of perturbing the expressions of our test suite. Algorithm 7 is our test harness: it picks the longest expression as the suspect unstable expression, perturbs that expression by replacing it with 5, 50, 100, 300, 500, and finally all expressions[1], and, for each set of variants, outputs the MD and CV of the results of executing each perturbed test. Because `Poly.c` and `Inv.c` each have a variable with randomly generated values, we ran both functions 1000 times each.

Each program in Figure 12 has two rows, one that reports the MD while the other reports the CV. After each program name, we give the total number of variants we generated. The longest expressions

[1] If the number of expressions chosen is less than the set of all equivalent expressions in $\mathbb{R}$ that subset is chosen uniformly at random.

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

(a) `Inv.c`

```c
double x = 1.0;
for (i=1; i<=6; i++) {
  x = 0.5 * (x + 2.0/x);
}
```

(b) `Newton.c`

```c
float x,z;
x = random(0,1);
z=x*x*x*x-4*x*x*x+6*x*x-4*x+1;
```

(c) `Poly.c`

```c
float a,b,c,d;
a=98765.0; b=1.0;
c=5.0e-8; d=5.0e-8;
float A=a*((b+c)+d);
```

(d) `exp.c`

```c
double alpha, NA, re, Z, L, Lp, A;
Z = 4; A = 26; re = 11.3;
alpha = 0.7; NA = 12.5;
L = log( 184.15 / pow(Z,1.0/3) );
Lp = log( 1194.0 / pow(Z,2.0/3) );
double X0=(4.0*alpha*re*re)
          *(NA/A)*(Z*Z*L + Z*Lp);
```

(e) `sample_run.c`

```c
float a, b, c, r1;
a=7; b=8686; c=2;
r1=(-b+sqrt(b*b-4*a*c))/(2*a);
```

(f) `root.c`

Figure 8: Salient features of the test suite.

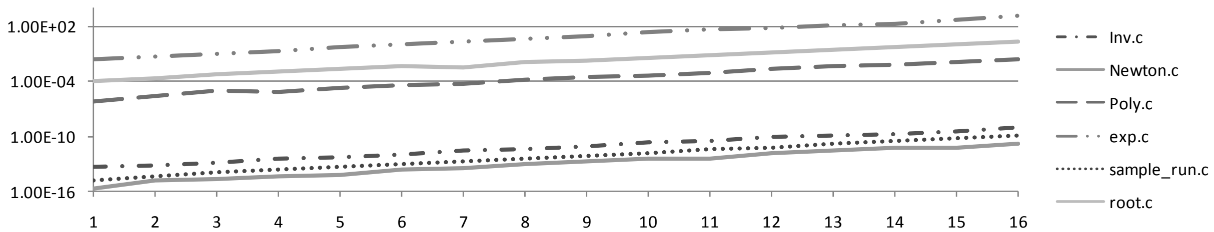


Figure 9: Dependence of MD under value perturbation on the length of the perturbed suffix.

in `Poly.c` and `sample_run.c` have more equivalent expressions in $\mathbb{R}$ than we could compute; (-) denotes this fact.

Under expression perturbation, the expressions in `Inv.c` and `Newton.c` are stable: their results do not vary at all; the expressions formed from rewriting their initial expression are also equivalent in floating-point values. Tests `root.c` and `sample_run.c` are also quite stable in the face of expression perturbation. The CV of both of them is less than 9.00E-9 and both of their MD are less than 8.00E-12. We cannot improve these programs much by rewriting the form of their expressions. `Poly.c` and `exp.c` are unstable. Enumerating all equivalent forms of `exp.c`'s longest expression generates only 168 different variants. Nonetheless, its MD exceeds 5.00E-3. `Poly.c` is unstable because on some runs its MD is more than 1.19E-7. Our results show that value and expression perturbations confirm and complement each other.

5.2 GSL

Here, we complement our earlier, small scale study of subjects drawn from the literature with a study of a large and widely-used numeric library, the GNU Scientific Library (GSL) [8]. We use our perturbation framework to evaluate the stability of its functions.

5.2.1 *Value Perturbation Results*

Figure 13 gives the maximum CV when perturbing GSL. We value-perturb all the functions in GSL except some that include static functions, because CIL does not support static functions. For our test suite, we choose the 48 single floating-point input and output functions in order of appearance in the GSL documentation. Many of these functions are not smooth. Some diverge at π. Since randomly generated inputs are unlikely to trigger such cases, we combined random and guided inputs to test these functions. We generated random inputs as follows: 30 inputs from $(-0.1, 0.1)$, 30 from $(-\pi, \pi)$, 30 from $(-100, 100)$, and 30 uniformly at random. For $\epsilon = 5E-10$ and $I = \{-e, -\pi, -2, -1, -\pi/2, 0, \pi/2, 1, 2, \pi, e\}$, we generated our guided inputs as follows: $\forall i \in I$, we selected ten inputs in the range of $(i - \epsilon, i + \epsilon)$, including i. Gathering the random and guided inputs, we generated 230 inputs in total. As above, we vary the perturbed suffix length from 1 to 16. We ran each function 1000 times on the generated set of inputs for each suffix length. Due to page constraints, we present only the maximum CV of each function with a suffix length of 5-bits in Figure 13[2].

Holding the function fixed, the CV results at different suffix lengths are similar. For example, the maximum CV across all suffix lengths for `gsl_sf_bessel_I1_scaled` is 2.12E-37. However, the CV of different functions are quite different: across all 48 functions, when perturbing 6-bits, the variance is 1.12E+266. The function `gsl_sf_erf` generates the maximum CV 1.11E-16 at input 1.52E+01 with 1-bit perturbation, while 1.07E-16 at input 7.81E+01 with 16-bits perturbation. Figure 13 shows the maximum CV of these functions when we perturb a 5-bit suffix. The horizontal axis is the name of the evaluated function; the vertical axis is the maximum CV in log scale. Some functions like `gsl_sf_bessel_j0` and `gsl_sf_bessel_K1` are especially sensitive: Their maximal CV values are 1.62E+131 and 1.64E+126 respectively. However, maximum CV of most of the functions in our test suite is quite low; 30 functions' maximum CV is negligible.

5.2.2 *Expression Perturbation Results*

We applied expression perturbation to those functions in Figure 13 whose CV exceeded 1E+60, using exactly the same inputs we gen-

[2]The full data set is available for download at `http://www.cs.ucdavis.edu/~su/GSLResult.tar.gz`.

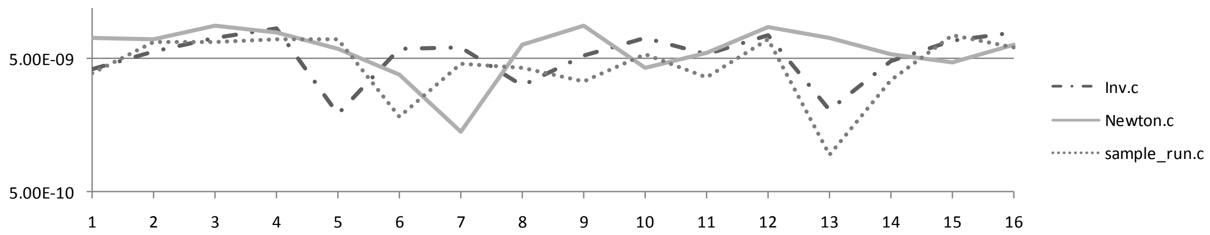


Figure 10: Dependence of CV under value perturbation on the length of the perturbed suffix (Stable programs).

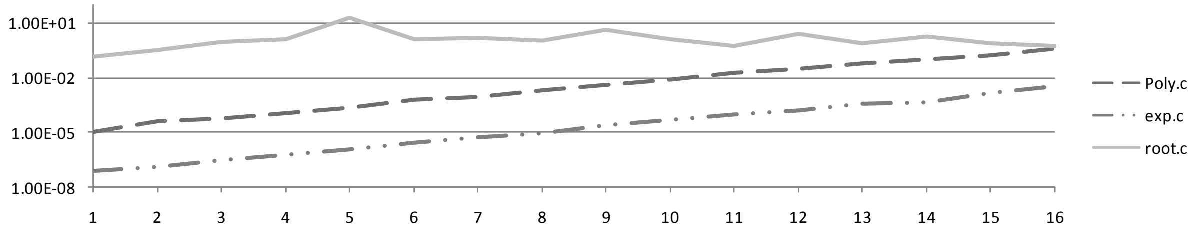


Figure 11: Dependence of CV under value perturbation on the length of the perturbed suffix (Unstable programs).

No. of Exp replaced		5	50	100	300	500	all
Inv.c(48)	(MD)	0	0	0	0	0	0
	(CV)	0	0	0	0	0	0
Newton.c(56)	(MD)	0	0	0	0	0	0
	(CV)	0	0	0	0	0	0
Poly.c(-)	(MD)	5.96E-08	1.19E-07	1.19E-07	1.19E-07	1.19E-07	-
	(CV)	5.455E-08	4.056E-08	3.599E-08	4.818E-08	4.815E-08	-
exp.c(168)	(MD)	7.81E-03	1.56E-02	1.56E-02	1.56E-02	1.56E-02	1.56E-02
	(CV)	3.64E-08	4.798E-08	5.227E-08	5.048E-08	5.048E-08	5.05E-08
sample_run.c(-)	(MD)	3.64E-12	7.28E-12	7.28E-12	7.28E-12	7.28E-12	-
	(CV)	7.184E-09	8.452E-09	8.783E-09	8.055E-09	8.179E-09	-
root.c(1920)	(MD)	2.27E-13	2.27E-13	2.27E-13	2.27E-13	2.27E-13	2.27E-13
	(CV)	6.782E-09	6.093E-09	6.086E-09	5.987E-09	5.903E-09	7.08E-09

Figure 12: MD and CV from expression perturbation.

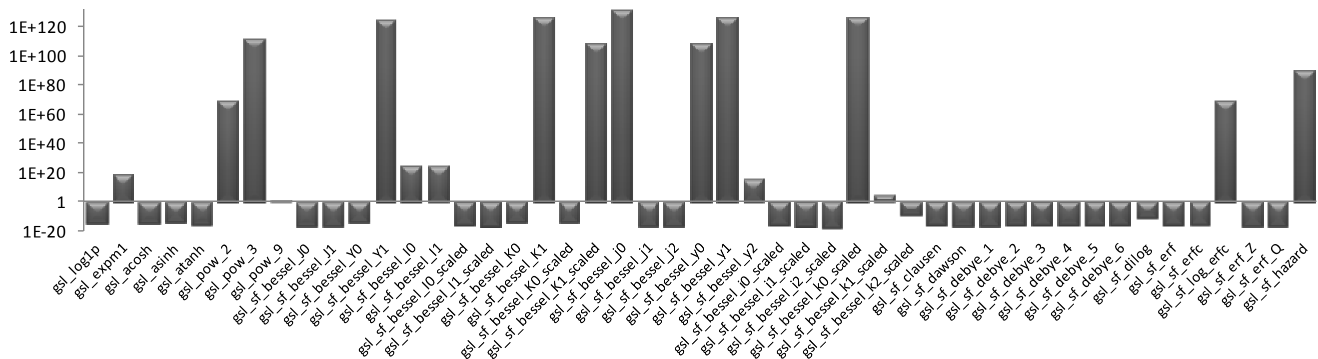


Figure 13: GSL maximum CV under value perturbation.

Function name	Max-MD	Max-CV	Input
gsl_pow_2	0	0	any
gsl_pow_3	0	0	any
gsl_sf_bessel_j0	1.26E+117	2.07E+233	-1.48E+26
gsl_sf_bessel_k0_scaled	0	0	any
gsl_sf_bessel_K1	1.42E-14	4.90E-29	8.15E-02
gsl_sf_bessel_K1_scaled	1.42E-14	4.90E-29	9.98E-03
gsl_sf_bessel_y0	1.11E-16	2.75E-33	9.98E-03
gsl_sf_bessel_y1	2.22E-16	8.24E-33	3.67E+01
gsl_sf_hazard	1.11E-16	2.75E-33	3.67E+01
gsl_sf_log_erfc	9.86E-32	8.06E-64	-1.00E+00
gsl_sf_bessel_Y1	4.90E+55	5.83E+110	1.54E-72

Figure 14: Max MD and CV for GSL expression perturbation.

erated during value perturbation. We used Algorithm 7 to generate a set of perturbed functions for each expression in each tested function, then generated the MD and CV of the results of running these sets of functions on our input set. Figure 14 reports the maximum from the resulting sets of MDs and CVs and also shows the input at which those maxima occurred. Three of the functions — `gsl_pow_2`, `gsl_pow_3`, and `gsl_sf_bessel_K1` — are quite stable. Two are strikingly unstable — `gsl_sf_bessel_Y1` and `gsl_sf_bessle_j0`. Judging from the magnitude of the reported statistics, the remaining functions appear stable, but may bear further investigation.

We tracked the suspected instability of `gsl_sf_bessle_Y1` to the expression `result->val = two_over_pi * lnterm * J1.val + (0.5 + c.val)/x`; and that of `gsl_sf_bessle_j0` to the expression `result->val = z * (1.0 + z*z * sin_cs_result.val);`. Expression perturbation has identified these two expressions as critical for further investigation.

5.3 Threats to Validity

Our technique is neither sound nor complete — it may miss critical inputs and is subject to both false positives and false negatives. False negatives are fundamental to testing, which has, nonetheless, proven to be a useful and practical technique for improving software. We believe that our technique will be similarly useful to the developers of numeric code. Our test suite is small, but representative and, in conjunction with our study of the GSL, the reported results are promising. Our testing harness selects inputs to each tested function uniformly from that function's domain and, for expression perturbation, it selects from among an expression's variants uniformly at random. Along these two dimensions, we obey the Monte Carlo constraint. However, our expression perturbation perturbs each floating-point expression in a program in isolation; we do not select subsets of a program's expressions and thus we do not test expression composition. Our expression perturbation technique assumes that an expression it rewrites is not already written in its most stable form; violations of this assumption will cause our technique to falsely classify such expressions as unstable. Finally, it can be difficult for a user, especially a novice, to interpret our results as there are no simple guidelines for knowing when an MD or CV is cause for concern. Here, we trust that having more data about the behavior of a program is better than less, and that the data we report will help bootstrap and refine developers' understanding of their programs and whether the data suggests a significant instability.

6. RELATED WORK

Much work has been done on roundoff error and stability analysis for numerical algorithms, but not for numerical implementations [12, 20, 21, 22, 28]. We survey the closely related work on the analysis of numerical software. Two threads underlie related work: 1) static analysis of numerical programs and 2) stochastic arithmetic based on perturbations.

Static Analysis of Numerical Programs In recent years, static analysis techniques have been developed to analyze numerical programs. We mention a few representative efforts here. Goubault [10] develops an abstract interpretation-based static analysis [4] to analyze errors introduced by the approximation of floating-point arithmetic. The work was later refined by Goubault and Putot [11] with a more precise abstract domain. Martel [16] presents a general concrete semantics for floating-point operations to explain the propagation of roundoff errors in a computation. In later work [17], Martel applies this concrete semantics and designed an static analysis for checking the stability of loops [12]. Martel develops a static analysis to optimize the numerical accuracy of expressions [18] and of general programs [19]. He exploits the same fact that we have in this work: numerical expressions equivalent over $\mathbb{R}$ may not be equivalent under floating-point semantics. Our work complements these static analyses by offering alternative, statistics-based analysis for validating numerical software.

Stochastic Arithmetic Another thread of related research concerns stochastic arithmetic. The representative work is CESTAC [3]. It performs transformations similar to our value perturbation, but for a different purpose. Instead of evaluating the robustness of numerical programs, CESTAC executes a program some number of times and uses the resulting mean to indicate the significant bits in the value. This technique has been incorporated into the CANDA tool [27]. Similar to the basic idea behind stochastic arithmetic, Parker et al. [24, 25] formalize and introduce the Monte Carlo Arithmetic (MCA) framework. This framework allows random rounding and unrounding to simulate computational error. Our value perturbation is similar to MCA. Based on MCA, the authors built their wonglediff tool [5], which changes the rounding mode of an FPU while a program is running. Unlike our work, it works on unmodified numeric programs, but is restricted to rounding modes and therefore does not perturb expressions and does not support general value perturbation. Our work is based on the general notion of perturbation. We extend existing work with expression perturbation and realize it in a practical tool. We also provide significant empirical evaluation to show our technique's practical benefits.

7. CONCLUSION AND FUTURE WORK

Numerical programs are difficult to get right, especially for application programmers who tend to think in terms of ideal real arithmetic instead of floating-point approximations on computers. We have developed a novel, practical framework to help programmers gain high-level knowledge about their scientific programs and warn of potential numeric errors in their code. We exploit the concept of perturbation in developing our framework, which consists of the complementary value and expression perturbations. Value perturbation uncovers intrinsic floating-point errors by randomly altering the least significant bits of computed values. Expression perturbation uncovers unstable expressions by statistically comparing an expression's mathematically equivalent forms. Our evaluation on various test programs and numerical library code shows the practical benefits of the proposed framework. We believe our technique offers programmers a useful utility in developing scientific applications.

There are a few interesting directions for future work. First, our expression perturbation works on individual expressions in a program. We plan to extend it to handle expression sequences or an

entire function. This capability will open up additional opportunities for perturbation and uncovering hidden errors. Second, we would like to explore test input generation techniques for numerical programs to focus on more relevant boundary values. Perturbing these values is more likely to reveal subtle errors. Third, perturbation is a general concept, and we plan to investigate other useful instantiations in addition to expression and value perturbations.

Acknowledgements

We would like to thank Zhaojun Bai, Eric Hyatt, and William Kahan for their helpful discussions and feedback. This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, and the U.S. Air Force under grant FA9550-07-1-0532. The authors at Nanjing University are supported by the National Natural Science Foundation of China (No.90818022 and No.60721002) and the National 863 High-Tech Program of China (No.2009AA01Z148). This material is based in part upon work supported by the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001, under the auspices of the Institute for Information Infrastructure Protection (I3P) research program. The I3P is managed by Dartmouth College. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Department of Homeland Security, the I3P, or Dartmouth College.

References

[1] A. Adare, S. Afanasiev, C. Aidala, N. N. Ajitanand, Y. Akiba, H. Al-Bataineh, J. Alexander, K. Aoki, L. Aphecetche, R. Armendariz, S. H. Aronson, J. Asai, E. C. Aschenauer, E. T. Atomssa, R. Averbeck, T. C. Awes, B. Azmoun, V. Babintsev, M. Bai, G. Baksay, L. Baksay, A. Baldisseri, K. N. Barish, P. D. Barnes, B. Bassalleck, A. T. Basye, and S. Bathe. Double-helicity dependence of jet properties from dihadrons in longitudinally polarized $p + p$ collisions at $\sqrt{s} = 200\ gev$. *Physical Review D*, 81(1):012002, Jan 2010.

[2] American NationalStandards Institute. IEEE standard for binary floating-point arithmetic, 1985.

[3] M.-C. Brunet and F. Chatelin. CESTAC, a tool for a stochastic round-off error analysis in scientific computing. *Numerical Mathematics and Applications*, pages 11–20, 1986.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 234–252, 1977.

[5] P. R. Eggert and D. S. Parker. Perturbing and evaluating numerical programs without recompilation: the wonglediff way. *Software Practice and Experience*, 35(4):313–322, 2005.

[6] G. Fishman. *Monte Carlo*. Springer, corrected edition, 2003.

[7] A. G. Frodesen, O. Skjeggestad, and H. Tøfte. *Probability and Statistics in Particle Physics*. Universitetsforlaget, 1979.

[8] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. *Gnu Scientific Library Reference Manual*. Network Theory Ltd., 1.2 edition, 2002.

[9] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991.

[10] E. Goubault. Static analyses of the precision of floating-point operations. In *Proceedings of the 8th International Static Analysis Symposium*, pages 234–259, 2001.

[11] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Proceedings of the 13th International Static Analysis Symposium*, pages 18–34, 2006.

[12] N. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics, 2002.

[13] L. Jiang and Z. Su. Osprey: a practical type system for validating dimensional unit correctness of c programs. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 262–271, 2006.

[14] W. Kahan. The improbability of probabilistic error analyses for numerical computations. First presented in 1995 in Hamburg at the third ICIAM Congress. http://www.cs.berkeley.edu/~wkahan/improber.pdf, 1996.

[15] D. E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[16] M. Martel. Propagation of roundoff errors in finite precision computations: a semantics approach. In *11th European Symposium on Programming*, pages 194–208, 2002.

[17] M. Martel. Static analysis of the numerical stability of loops. In *Proceedings of the 9th International Static Analysis Symposium*, pages 133–150, 2002.

[18] M. Martel. Semantics-Based transformation of arithmetic expressions. In *Proceedings of the 14th International Static Analysis Symposium*, pages 298–314, 2007.

[19] M. Martel. Program transformation for numerical precision. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 101–110, 2009.

[20] W. Miller. Toward mechanical verification of properties of roundoff error propagation. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 50–58, 1973.

[21] W. Miller. Software for roundoff analysis. *ACM Trans. Math. Softw.*, 1(2):108–128, 1975.

[22] W. Miller and D. Spooner. Software for roundoff analysis, ii. *ACM Trans. Math. Softw.*, 4(4):369–387, 1978.

[23] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, 2002.

[24] D. Parker, B. Pierce, and P. Eggert. Monte carlo arithmetic: how to gamble with floating point and win. *Computing in Science & Engineering*, 2(4):58–68, 2000.

[25] D. S. Parker, P. R. Eggert, and B. Pierce. Monte Carlo Arithmetic: a framework for the statistical analysis of roundoff error, 1997.

[26] D. Quinlan. ROSE: Compiler support for object-oriented frameworks. In *Proceedings of Conference on Parallel Compilers (CPC2000)*, pages 215–226, 2000.

[27] J. Vignes. A stochastic arithmetic for reliable scientific computation. *Mathematics and Computers in Simulation*, 30(6):481–491, 1988.

[28] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. HMSO, London, UK, 1963.