Automated Backward Error Analysis for Numerical Code

Zhoulai Fu Zhaojun Bai Zhendong Su
Department of Computer Science, University of California, Davis, USA
{zlfu, zbai, su}@ucdavis.edu

Abstract

Numerical code uses floating-point arithmetic and necessarily suffers from roundoff and truncation errors. Error analysis is the process to quantify such uncertainty. *Forward error analysis* and *backward error analysis* are two popular paradigms of error analysis. Forward error analysis is intuitive, and has been explored and automated by the programming languages (PL) community. In contrast, although backward error analysis is fundamental for numerical stability and is preferred by numerical analysts, it is less known and unexplored by the PL community.

To fill this gap, this paper presents an *automated backward error analysis* for numerical code to empower both numerical analysts and application developers. In addition, we use the computed backward error results to compute the *condition number*, an important quantity recognized by numerical analysts for measuring a function's sensitivity to errors in the input and finite precision arithmetic. Experimental results on Intel x87 FPU instructions and widely-used GNU C Library functions demonstrate that our analysis is effective at analyzing the accuracy of floating-point programs.

Categories and Subject Descriptors D.1.2 [*Automatic programming*]: Programming transformation; G.1.0 [*General*]: Error analysis, Conditioning; G.4 [*Mathematical Software*]: Algorithm design and analysis

General Terms Reliability, Algorithm

Keywords Floating point, backward error, condition number, mathematical optimization

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA

http://dx.doi.org/10.1145/2814270.2814317

1. Introduction

> *Floating point computation is by nature inexact, and it is not difficult to misuse it so that the computed answers consist almost entirely of "noise".*
>
> — D. Knuth, *The Art of Computer Programming, [31]*

Numerical error is inherent in machine computation. This is particularly true when we talk about floating-point programs. Admittedly, the goal of floating-point programs is rarely to compute the exact answer, but a result that is sufficiently accurate. The ability to measure the accuracy of numerical programs is, therefore, essential.

There are two ways to measure numerical accuracy. One is called *forward error analysis*, which is to directly measure the difference between the computed solution on a finite-precision arithmetic and the exact solution (usually simulated by a high-precision arithmetic, or an oracle solution). The forward error analysis has been extensively studied and drawn on almost all modern PL techniques, such as static analysis [10], formal deduction [11], and symbolic execution [8].

For numerical analysts, a more appealing paradigm for analyzing numerical code is to use *backward error analysis* (BEA). BEA has been successfully applied to *manually* analyze many fundamental algorithms [26]. However, BEA is much less known in the PL community. It seems that "numerical analysts failed to influence computer software and hardware in the way they should" according to the Turing lecture delivered by J. H. Wilkinson [45].

In this work, our goal is two-fold. First, we introduce BEA from applied mathematicians and numerical analysts to the PL community. Second, at the technical level, we present novel techniques to automate BEA to benefit both numerical analysts — who perform BEA largely on paper-and-pencil — and application developers — who will now understand the accuracy and stability of their numerical code via au automated approach.

Backward Error. Consider the problem to be solved by a mathematical function f that maps the input data x to the solution $f(x)$. Let $\hat{f}$ be the numerical code that implements f. The result of the implementation, $\hat{f}(x)$, will usually deviate from the mathematical $f(x)$. A primary concern in using $\hat{f}(x)$ as an implementation of f is to estimate the *relative forward*

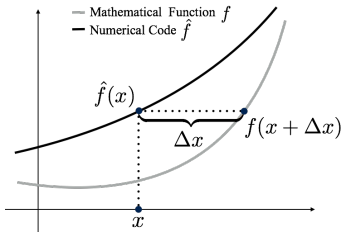

Figure 1: Illustration of backward error.

error, denoted by F:

$$F \triangleq \left| \frac{\hat{f}(x) - f(x)}{f(x)} \right|. \tag{1}$$

Rather than computing F, the BEA approach attempts to view $\hat{f}(x)$ as the result of f with a slightly perturbed data at x, *i.e.*,

$$\hat{f}(x) \simeq f(x+\Delta x). \tag{2}$$

We call such Δx *backward error* (Fig. 1).

Why do we need to concern with backward error? A major benefit is that it allows for a simple yet very useful separation of concerns in understanding numerical accuracy. Let

$$\delta = \Delta x / x \tag{3}$$

denote the relative perturbation of Δx with respect to x. Assuming that f is smooth in a neighborhood of x and δ is small, then by (1) and Taylor expansion, we have

$$\begin{aligned} F &= \left| \frac{f(x+\delta \cdot x) - f(x)}{f(x)} \right| \\ &\approx |\delta| \cdot \left| \frac{x \cdot f'(x)}{f(x)} \right| + \mathscr{O}(\delta^2). \end{aligned} \tag{4}$$

Then we can divide the question of forward error estimation into two:

- Backward error $B = |\delta|$ that is implementation-dependent (B depends on $\hat{f}$), and
- Condition number $C = \left| \frac{x \cdot f'(x)}{f(x)} \right|$, which is inherent in the mathematical problem to solve and independent of the implementation $\hat{f}$.

Thus, we can summarize the power of BEA as

$$F \approx B \times C. \tag{5}$$

The knowledge of backward error allows us to track down the inaccuracy, *i.e.*, large forward error, to either B or C. If C is large, the problem is called *ill-conditioned*, and theoretically it is very difficult to cure the inaccuracy. If C is relatively small, we should have a large backward error causing the inaccuracy. In the latter case, efforts should be made to seek a better implementation.

Table 1: Understanding forward error via backward error.

F	B	Analysis
Small	Large	Accuracy insensitive to implementation
Small	Small	Good implementation
Large	Small	Ill-conditioned problem
Large	Large	Reducing backward error *may* help

Along the same line, using backward error also allows us to optimize the program. Let us consider the case when the forward error is already small, but the backward error is large. We must have a subtle condition number. In this case, the forward error is insensitive to the implementation, and there is little benefit to devise a fine-tuned or highly precise numerical code; in other words, we may use a simpler numerical implementation without sacrificing much accuracy.

Tab. 1 shows the different configurations and their conclusions drawn from a backward error analysis.

Condition Number. Another utility of computing backward error is to estimate the *condition number*. If we simultaneously simulate F and B, the condition number C can be obtained immediately as F/B.

Estimating the condition number is very important in understanding the accuracy of floating point programs. Rewrite (1) to the following equation:

$$\log F = \log B + \log C. \tag{6}$$

Consider $\log F$ as the inaccuracy measured in terms of its significant digits. Eq. (6) means that we may lose up to $\log C$ digits of accuracy on top of what would be lost due to loss of precision from arithmetic methods (a rule of *significance arithmetic* [22]). Thus, if the magnitude of B reflects how bad our implementation is, an estimation of C allows us to quantify the inaccuracy that is "born with" the problem that we want to solve. In Sect. 2 and 4, we show the technique of condition number estimation to analyze an inaccuracy issue posed on the `fsin` instruction of Intel.

Note that computing condition number is commonly regarded as a more difficult problem than solving the original problem[1], *e.g.*, the condition number for a linear system $AX = B$ involves computing the inverse of A, which is known to be a harder problem than solving $AX = B$. As Higham put it [26] (p.29): "The issue of conditioning and stability play a role in all discipline in which finite precision computation is performed, but the understanding of these issues is less well developed in some disciplines than others." In the field of programming languages, the theory of condition number is unfamiliar. We give a practical and systematic way to esti-

[1] Computing condition number is generally more difficult unless certain necessary quantities are already pre-calculated, for example if the LU factorization of a matrix has been computed, then the condition number estimator is an order less than computing the LU factorization. See [26], Chap. 15.

mate condition number that should benefit a wide range of researchers and application developers.

Contributions. This paper describes the design and implementation of an automated BEA framework for floating-point programs. We develop two techniques to approach backward error analysis. One is to focus on how to understand the detailed characteristics of numerical code at a single point and within a range of points of interest, called *local BEA*; another one is to estimate the backward error across an input range, called *global BEA*. The BEA techniques not only provide us an insight into numerical implementation, but also contributes to another, less explored body of research — estimating condition numbers, which is a key to understand the inaccuracy inherent in the underlying mathematical problem. Our contributions follow:

- This is the first work to successfully automate the process of computing backward error. We introduce novel techniques to make the analysis systematic and effective.
- Using the computed backward error, we estimate the condition number and show how it helps understand the source of floating-point inaccuracy in some well-known examples.

Outline. We begin by describing an overview of our approach in Sect. 2. The core algorithms for computing backward error are introduced in Sect. 3. Then, we explain some important implementation details, experimental design and present the evaluation in Sect. 4. Finally, we discuss related work in Sect. 5 and conclude in Sect. 6.

Notation. The set of real and integer numbers are denoted by $\mathbb{R}$ and $\mathbb{Z}$ respectively. For two real numbers a and b, the usage "aEb" means $a*10^b$. A function can be expressed as a lambda expression, *e.g.*, $\lambda x.x*x$ for the square function.

Given $X \subseteq \mathbb{R}$, called *search space*, and $\mathscr{E} : X \to \mathbb{R}$, called *objective function* (or *energy function*), we call x^* a *local minimal point*, if there exists a *neighborhood* of x^*, namely $\{x \in X \mid |x-x^*| \leq d\}$ for some $d > 0$, such that for all x in the neighborhood, $\mathscr{E}(x^*) \leq \mathscr{E}(x)$. The value of $\mathscr{E}(x^*)$ is called *local minimum*. If $\mathscr{E}(x^*) \leq \mathscr{E}(x)$ for all $x \in X$, we call x^* a *global minimum point* and $\mathscr{E}(x^*)$ the *global minimum*.

2. Overview and Two Examples

We let $\hat{f}$ denote a numerical program with scalar input and output. The *mathematical function* that $\hat{f}$ attempts to compute is denoted by f.

Unlike previous efforts that study backward error à la pencil-and-paper, our approach attempts to compute backward error fully automatically. The kernel of our approach is structured into the following components (Fig. 2):

- *Local BEA* determines the backward error at single inputs. Because f is only conceptional, we lift $\hat{f}$ to higher precision to simulate f. This first step involves a source-level transformation commonly used in floating-point program analysis [7, 9, 13]. We shall then regard f as the exact solution. The essence of solving local BEA is to, for a given input x, find the smallest $|\delta|$ such that the perturbed $x+\delta \cdot x$ applied on f comes close to $\hat{f}(x)$. The derived mapping that associates input x with the obtained smallest $|\delta|$ defines the backward error function $B(x)$.
- *Global BEA* uses $B(x)$ as a black-box function and estimates the *maximal* backward error for a given search space. We use global BEA results to quantify the worst-case inaccuracy of the implementation over the search space. We employ classic *Monte Carlo Markov Chain* (MCMC) techniques for solving global BEA.
- As an application, we apply our BEA techniques to estimate condition numbers of numerical problems, using

$$C(x) \triangleq \frac{F(x)}{B(x)} \tag{7}$$

where F is the forward error as defined in (1). Similar to our global BEA approach, we use the MCMC engine to estimate the largest condition number for a given range.

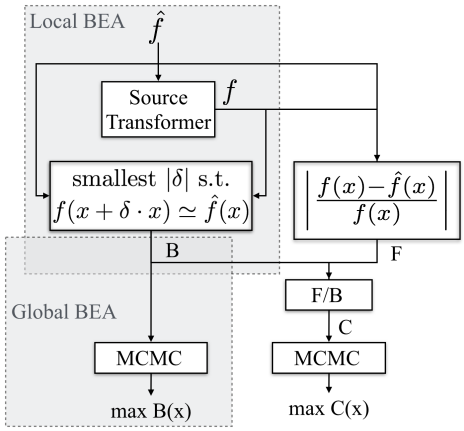

Figure 2: Overview of our approach. $\hat{f}$: numerical code under analysis, f: the transformed program of a higher precision than $\hat{f}$, B: backward error, F: forward error. The two shadowed parts represent local BEA and global BEA.

Below, we give two examples. Our goal is to study two different causes of inaccuracy. The first example shows an implementation problem due to large backward error, whereas the second exhibits inaccuracy from the result of a large condition number. Through the examples, we show that some inaccuracy issues can be easily fixed, and some may not, and an appropriate way to understand the difference is through backward error analysis.

Example 1. Suppose we want to compute

$$f(x) = \sqrt{x+1} - 1 \tag{8}$$

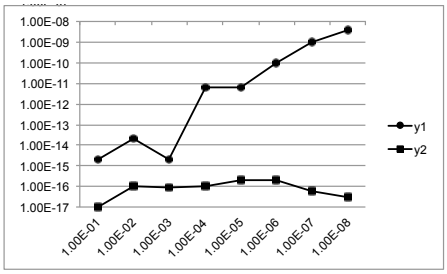

Figure 3: Backward error for `y1` and `y2` at $x = 1\text{E-}i$ where $i = 1 \ldots 8$.

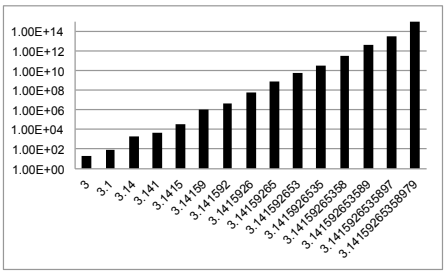

Figure 4: Condition number of `fsin` near π (corresponding to our experimental results reported in Tab. 7).

with a small input $x > 0$. Below is how most people would likely implement (8)

```
double y1 (double x)
{return sqrt(x + 1) - 1;}
```

However, for small x close to 0, this implementation suffers from *cancellation* error when performing the subtraction.

Consider, for example, input x = 2E-15. We have y1/(x) = 8.8817E-16,[2] yet the correct answer should be f(x)=1.000E-15. The forward error is $|(f(x) - y1(x))/f(x)| \simeq 0.12$. Given the forward error alone, however, we do not know the reason for it or whether it can be avoided. The inaccuracy can be attributed to either bad implementation or ill-conditioned problem. BEA allows us to make this distinction.

If we compute the backward error, *i.e.*, the smallest $|\delta|$ such that

$$\sqrt{(x + \delta \cdot x) + 1} - 1 = y1(x), \tag{9}$$

then we can obtain the backward error 0.112.

As discussed in Sect. 1, large backward error indicates the possibility in improving the implementation. In fact, the backward error of y1 can be reduced by using the algebraically equivalent y2 below:

```
double y2 (double x)
{return x / (sqrt(x + 1) + 1);}
```

We use our approach to compute the backward errors of y1 and y2. In Fig. 3, we illustrate the large backward error of y1 versus the much smaller backward of y2. With a sequence of input x closer and closer to 0, the backward error of y1 quickly goes above machine epsilon, while y2 is much more stable with backward errors in the order of 1E-17 to 1E-16.

Example 2. This is an ill-conditioned problem from Intel FPU, where both forward and backward errors are large, and finding accurate alternative is much harder due to the inherent large condition number.

In a recent blog, Google engineer B. Dawson pointed out [2] some accuracy issues regarding how the `fsin` instruction is described in Intel's documentation [27]. Intel acknowledged the problem and announced [3] that they would clarify and improve the documentation.

[2] Evaluated on an x86_64 OS, with the default optimization options of LLVM 5.1 (clang-503.0.40).

Dawson attempted to approximate π by $\sin(\hat{\pi}) + \hat{\pi}$, where $\hat{\pi}$ is an initial estimate of π. The rationale behind this approach should be clear: If $\hat{\pi} = \pi + d$ where d is small, by Taylor expansion we have

$$\sin(\hat{\pi}) = -\sin(d) = -d + O(d^3). \tag{10}$$

Therefore, it is expected that

$$\sin(\hat{\pi}) + \hat{\pi} = (-d + \mathcal{O}(d^3)) + (\pi + d) = \pi + \mathcal{O}(d^3). \tag{11}$$

The problem is that accurately computing $\sin(\hat{\pi})$ is difficult when $\hat{\pi}$ is close to π. In fact, to approximate π by $\sin(\hat{\pi}) + \hat{\pi}$ is theoretically possible but problematic in practice. To see this, we give the experimental results (Fig. 4) on our estimated condition numbers for a sequence of inputs increasingly close to π.

From the figure, it can be seen that the condition number increases significantly when x comes closer and closer to π. For example, if $\hat{\pi} = 3.1415926535897$, we have the condition number on the order of 1E+13, meaning a precision loss of 13 decimal digits intrinsic to the problem on top of the inaccuracy from the implementation (see Eq. 6). Because the condition number is large, we may not blame Intel for their implementation of `fsin`, but the problem raised by Dawson – his algorithm involves computing `fsin` near π, which is an ill-conditioned problem. That said, Intel now uses glibc's software implementation of `sin` rather than an FPU instruction. The current glibc's `sin` implementation relies on a lookup table for computing certain values of `sin`.

Not only our approach can compute condition numbers for given inputs, but it also can automatically detect input points where the condition number is large. In fact, our experimental results show that `fsin` has a large condition number for every $k * \pi$, where k is a positive integer (Tab. 2).

Along the same line, using our approach we can find that the condition number of `fsin` becomes large for large input x. As a result, it is also difficult to compute accurately `fsin` with large x. In fact, it has been reported that Intel's `fsin` function does not compute correctly for large input x [4]. Tab. 3 shows our analysis for `fsin` 10^k, where $k \in [-4, 5] \cap \mathbb{Z}$.

Table 2: Analysis of `fsin` at $[k\pi-1, k\pi+1]$. x^*: maximum point, C^*: maximum condition number, T: time in seconds.

Range	x^*	C^*	T (s)
$[\pi-1, \pi+1]$	3.1415914E+00	4.7010279E+09	213.29
$[2\pi-1, 2\pi+1]$	6.2831941E+00	3.0507019E+09	213.06
$[3\pi-1, 3\pi+1]$	9.4247772E+00	1.9457442E+10	998.72
$[4\pi-1, 4\pi+1]$	1.2566332E+01	2.5810622E+10	218.46

Table 3: Analysis of `fsin` for $x = 10^k$, where $k \in [-4,5] \cap \mathbb{Z}$.

x	F	B	C
0.0001	5.10E-17	5.10E-17	1.00E+00
0.001	5.67E-18	5.67E-18	1.00E+00
0.01	4.27E-17	4.27E-17	1.00E+00
0.1	3.09E-17	3.10E-17	9.97E-01
1	2.11E-18	3.29E-18	6.42E-01
10	7.16E-17	4.64E-18	1.54E+01
100	6.03E-18	3.54E-20	1.70E+02
1000	4.68E-17	6.88E-20	6.80E+02
10000	3.84E-17	1.23E-21	3.12E+04
100000	3.54E-15	1.27E-21	2.80E+06

3. Approach

This section presents the theoretical underpinning of our approach, which will be formalized in a one-dimensional context. We start by introducing two components that we assume available to us.

Let f be a continuous function on interval $[b,e]$. Assume that we have a procedure `LM` with the signature

$$\texttt{LM}(f,b,e). \tag{12}$$

The procedure attempts to find a local minimal point $x^* \in (b,e)$, or its endpoints if no such minimal points can be found in (b,e). We assume the following property on `LM`[3]

A1. Procedure `LM` always terminates and returns $x^* \in [b,e]$ that is a local minimal point of f on $[b,e]$.

Another component is the root-finding procedure with the following signature

$$\texttt{RF}(f,b,e) \tag{13}$$

where f holds opposite signs at the interval endpoints, *i.e.*, $f(b) * f(e) < 0$. In numerical analysis, we call such procedure a "bracket root-finding procedure". Under the condition that f is continuous on $[b,e]$, the root-finding procedure is guaranteed to locate a zero in $[b,e]$ within finite steps[4]. A large number of implementations exist [36] for this root-finding procedure. One of the simplest, for example, may be the bisection root-finding procedure which tries to find a new pair of endpoints $[a_n, b_n]$ such that $|b_n - a_n|$ is half of the distance of the endpoints from the previous step. Thus, we assume

A2. If $f(b) * f(e) < 0$, then $\texttt{RF}(f,b,e)$ terminates and $f(\texttt{RF}(f,b,e)) = 0$.

3.1 Local BEA

We define relative backward error with two parameters of tolerance *xtol* and *ftol*.

Definition 1. *Given a numerical program $\hat{f}$, the corresponding mathematical function f and input $x \in \text{dom}(\hat{f})$, local BEA is defined as the following* mathematical optimization *(MO) [47] problem:*

$$\begin{aligned} &\textit{minimize} && |\delta| \\ &\textit{subject to} && |f(x+\delta\cdot x) - \hat{f}(x)| \le \textit{ftol} \\ & && |\delta| \le \textit{xtol} \end{aligned} \tag{14}$$

The mapping that associates input x and the solution to (14)*, $|\delta^*|$, is denoted by the function*

$$B(x) = |\delta^*| \tag{15}$$

We call $B(x)$ the relative backward error *at x, or* backward error *for short.*

Compared to (14), the perturbation Δx in (1) is expressed by x multiplied by a perturbation factor δ, the closeness parameterized by *ftol*, and the search space bounded by *xtol*.

We first present an outline of how we compute $B(x)$. Fix x, and let Φ_x denote the auxiliary function

$$\Phi_x(\delta) \triangleq \left| f(x+\delta\cdot x) - \hat{f}(x) \right|. \tag{16}$$

In a typical setting, we require that Φ_x is *well-behaved*[5] in the sense that

A3. Φ_x is continuous on $[-\textit{xtol}, \textit{xtol}]$.
A4. Φ_x has a moderate number of minima on $[-\textit{xtol}, \textit{xtol}]$ (*i.e.*, Φ_x should not oscillate a huge number of times).

Following the definition, if $\Phi_x(0) \le \textit{ftol}$, we immediately have $B(x) = 0$. Without loss of generality, we assume

$$\Phi_x(0) > \textit{ftol} \tag{17}$$

Imagine that we draw Φ_x as the curve illustrated in Fig. 5. The shadowed area denotes $\{(x,y) \mid |x| \le \textit{xtol}, y \in [0, \textit{ftol}]\}$. Computing $B(x)$ actually amounts to finding the intersection of the Φ_x curve and the upper boundary of the shadowed area.

[3] In calculus, the image of a continuous function on a bounded close set is necessarily bounded and closed (or more generally, compactness is preserved under a continuous map [38]).

[4] The root exists because of the intermediate value theorem [38].

[5] It is common practice in the numerical analysis literature to assume that the function under study is well-behaved in some sense. The study on pathological functions is a separate problem beyond the scope of this paper.

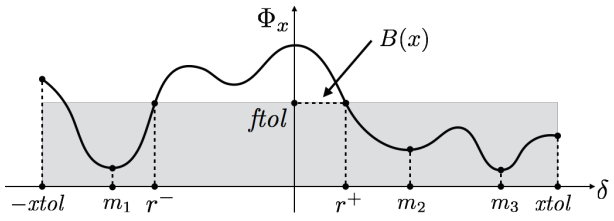

Figure 5: Illustration of the local BEA algorithm.

Lemma 1. *If δ^* is the solution to the MO problem (14), then $\Phi_x(\delta^*) = ftol$.*

To prove the lemma, we only need to exclude the possibility of $\Phi_x(\delta^*) < ftol$. Following **A3**, if $\Phi_x(\delta^*) < ftol$, we can find $\widetilde{\delta^*}$ such that $|\widetilde{\delta^*}| < |\delta^*|$ and $\Phi_x(\widetilde{\delta^*}) < ftol$. This contradicts the definition of backward error. As a direct corollary of Lem. 1, we have

$$B(x) = \min\{|\delta| \mid \Phi_x(\delta) = ftol, |\delta| \leq ftol\} \tag{18}$$

Our approach computes $B(x)$ in three high-level steps:

S1. Compute the smallest root $r^+ > 0$ s.t. $\Phi_x(r^+) = ftol$.
S2. Compute the largest root $r^- < 0$ s.t. $\Phi_x(r^-) = ftol$.
S3. Return $\min(|r^+|, |r^-|)$ as $B(x)$.

Now let us focus on **S1**. Although formulated as a root-finding step, traditional root-finding procedure is not enough here, which stops whenever an arbitrary root is found. We handle **S1** with two sub-steps:

S1a. First, we compute the smallest local minimal point $lm^+ \in [0, xtol]$ s.t. $\Phi_x(lm^+) \leq ftol$.
S1b. Then, we compute $r^+ \in [0, lm^+]$ s.t. $\Phi_x(r^+) = ftol$ using traditional root-finding procedure.

In Fig. 5, the desired results for **S1a** and **S1b** are marked m_2 and r^+ respectively. We justify why **S1** can be done with the two sub-steps using the following lemma.

Lemma 2. *Let lm^+ be*

$$\min\{m > 0 \mid m \text{ is a local minimal point of } \Phi_x, \Phi_x(m) \leq ftol\}.$$

Then, there exists a single $r^+ \in [0, lm^+]$ such that $\Phi_x(r^+) = ftol$.

Proof. The existence of r^+ follows directly from **A2**, because $\Phi_x(0) > ftol$, $\Phi_x(lm^+) \leq ftol$ and Φ_x is continuous (**A1**). We conduct the proof by contradiction by assuming that

H: $\exists r_1, r_2$ *s.t.* $0 < r_1 < r_2 \leq lm^+$ and $\Phi_x(r_1) = \Phi_x(r_2) = ftol$.

Since Φ_x is continuous on $[r_1, r_2]$, we are guaranteed to have a maximal point $z \in [r_1, r_2]$ *s.t.* $\Phi_x(z) \geq \Phi(r)$ for all $r \in [r_1, r_2]$. Because $\Phi_x(z) \geq ftol$, we only have two cases to consider: (1) If $\Phi_x(z) = ftol$, then we have $\Phi_x(r) = ftol$ for all $r \in [r_1, r_2]$. Therefore, each r in the range is a local minimal point of Φ_x, contradicting with the definition of lm^+; and (2) If $\Phi_x(z) > ftol$, we have a minimum bracket (o, r_1, z), *i.e.*, $0 < r_1 < z$ *s.t.* $\Phi_x(r_1) < \Phi_x(0)$ and $\Phi_x(r_1) < \Phi_x(z)$. Thus, there must exist a local minimal point in $[0, z]$. Because $z < r_2$ (otherwise $\Phi_x(r_2) > ftol$), and $z_2 \leq lm^+$, we have actually found a local minimal point that is strictly smaller than lm^+, contradicting with the definition of lm^+. Combining cases (1) and (2), we conclude that (H) is false. □

Similarly, we achieve **S2** by the two sub-steps below:

S2a. Compute lm^-, *i.e.*

$$\min\{m < 0 \mid m \text{ is a local min. point of } \Phi_x, \Phi_x(m) \leq ftol\}.$$

S2b. Compute $r^- \in [lm^-, 0]$ s.t. $\Phi_x(r^-) = ftol$.

In Fig. 5, the desired results for **S2a** and **S2b** are m_1 and r^- respectively.

Our approach is summarized in Algo. 1. In the beginning, the algorithm returns 0 if already $\Phi_x(0) \leq ftol$ (Lines 1-2). The overall loop (Lines 3-17) implements **S1** and **S2**. Line 18 corresponds to **S3**.

We initialize δ to the boundary of the search space, $xtol$ or $-xtol$ (Line 4). Variable lm is used to return lm^+ or lm^-.

The loop at Lines 7-13 corresponds to **S1a** and **S2a** (with loop index $i = 1$ or $i = 2$ respectively). At line 10, lm is updated whenever we have a better, *i.e.*, smaller δ *s.t.* $\Phi_x(\delta) \leq ftol$. At line 11, we use a contractor coefficient cc for accelerating the procedure. If δ_n denotes δ obtained for the n-th iteration, we have

$$\delta_{n+1} \leq cc * \delta_n. \tag{19}$$

Therefore, the program is guaranteed to terminate if cc is set strictly smaller than 1. The loop continues unless the iteration bound is reached, or the local minimization hits the boundary of the search space (Line 13). Lines 14-17 correspond to steps **S1b** and **S2b**.

3.2 Global BEA

Global backward error analysis aims at finding the maximum of the backward error within a user-defined range. Global BEA allows us to automatically *detect* implementation issues quantified by backward error, as opposed to local BEA, which computes backward locally, and confirms or refutes the implementation issue at the given points. Note that the local backward error computed earlier is now regarded as a black-box function. In the one-dimensional case, we are concerned with finding its maximum within an interval $[b, e]$. Global BEA attempts to give a tight estimate of

$$\max_{x \in [b,e]} B(x). \tag{20}$$

We use *Monte Carlo Markov Chain* (MCMC) sampling [6] to solve (20). MCMC is a random sampling technique used to simulate a target distribution. For example, if we have the target distribution of coin tossing, with 0.5 probability

Algorithm 1: Local BEA

Input:

x	Input point
$\hat{f}$	Numerical code
f	Mathematical function
$xtol$	Bound of the search space
$ftol$	Tolerance controlling the closeness between $\hat{f}$ and f
cc	Contraction coefficient ($0 < cc \leq 1$)
$iter\_local$	Maximum iteration times of `LM`
`LM`	Local minimization procedure
`RF`	Root-finding procedure

Output: An estimation of $B(x)$

1 Let $\Phi_x = \lambda\delta.|f(x+\delta\cdot x) - \hat{f}(x)|$
2 **if** $\Phi_x(0) \leq ftol$ **then return** 0
3 **for** $i \in \{1,2\}$ **do**
  /* $i=1$ (resp. $i=2$) deals with the search space $[0, xtol]$ (resp. $[-xtol, 0]$) */
4 Let $\delta = \begin{cases} xtol & \text{if } i = 1 \\ -xtol & \text{otherwise} \end{cases}$
5 Let $lm = \delta$
6 Let $iter = iter\_local$
7 **repeat**
    /* Loop Invariant: $\Phi_x(lm) \leq ftol$ */
8   Let $\delta_{old} = \delta$
9   Let $\delta = \begin{cases} \texttt{LM}(\Phi_x, 0, \delta) & \text{if } i == 1 \\ \texttt{LM}(\Phi_x, \delta, 0) & \text{otherwise} \end{cases}$
10   **if** $\Phi(\delta) \leq ftol$ **then** $lm = \delta$
    /* Contracting the search bound for termination */
11   Let $\delta = \delta * cc$
12   Let $iter = iter - 1$
13 **until** $iter == 0$ *or* $\delta == \delta_{old}$
14 **if** $i == 1$ **then**
    /* lm is the smallest local minimal point $lm \in [0, xtol]$ such that $\Phi_x(lm) \leq ftol$ */
15   Let $r^+ = \texttt{RF}(\lambda\delta.\Phi_x(\delta) - ftol, 0, lm)$
16 **else**
    /* lm is the largest local minimal point $lm \in [-xtol, 0]$ such that $\Phi_x(lm) \leq ftol$ */
17   Let $r^- = \texttt{RF}(\lambda\delta.\Phi_x(\delta) - ftol, lm, 0)$
18 **return** $\min(|r^+|, |r^-|)$

for having head or tail. An MCMC sampling is a sequence of random variables $x_1, \ldots, x_n$, such that the probability of x_n being "head", denoted by P_n, converges to 0.5, *i.e.*, $\lim_{n\to\infty} P_n = 0.5$. The fundamental fact regarding MCMC sampling can be summarized as the theorem below [6]. For brevity, we only consider discrete-valued probability.

Theorem 1. *Let x be a random variable. Let A be an enumerable set of the possible values of x. Given a target distribution expressed by a density function $P(x = a)$ for $a \in A$. Let $x_1, \ldots, x_n$ be an MCMC sampling sequence (which is a Markov chain), and $P(x_i = a)$ be the density function regarding each x_i. We have*

$$\lim_{i\to+\infty} P(x_i = a) = P(x = a). \tag{21}$$

In short, the MCMC sampling follows the target distribution in the limit.

Why do we adopt MCMC approach for global BEA? There are two advantages. First, the search space in our problem setting is a subset of floating-point values. Even in the one-dimensional case, a very small interval contains a huge number of floating-point numbers. The MCMC approach is known as an effective technique in dealing with large search spaces. Second, our objective function, $B(x)$, is not smooth or not even continuous. In fact, even if both the numerical code $\hat{f}$ and the mathematical function f are continuous, $B(x)$ as in Def. 1 may still be discontinuous. In addition, $B(x)$ in practice has a large number of fluctuations, which makes many deterministic global optimization problems, such as convex analysis, interval analysis, and linear/non-linear programming [17], only effective when the objective functions are well-behaved, inappropriate. That said, other stochastic techniques, *e.g.*, genetic programming, may also be used for our problem setting. We have not experimented genetic programming (which we leave for future work) mainly because MCMC techniques have a stronger mathematical foundation.

Metropolis-Hasting is a commonly used and general MCMC sampling technique. Let $\mathscr{E}(x)$ be an energy distribution and x the current sampled point. The next Metropolis-Hasting sampling, denoted by

$$\texttt{Metropolis-Hasting}(\mathscr{E}, x) \tag{22}$$

is processed in two steps. First, we randomly *propose* a new sample x' from the current sample x. The distribution of the proposal has to meet some requirements,[6] which we do not detail here. A common choice of the proposed x', for example, is to choose x' randomly following a Gaussian distribution centered at x.

The second phase is to decide whether we should *accept* x' or not. Following Metropolis-Hasting algorithm, we use $\mathscr{E}(x')/\mathscr{E}(x)$ as the *acceptance ratio*: If $\mathscr{E}(x') > \mathscr{E}(x)$, then the proposed x' will be accepted. Otherwise, x' may still be accepted, but only with the probability of $\mathscr{E}(x')/\mathscr{E}(x)$.

The pseudocode of the sampling process is shown in Algo. 2, lines 12-20. To fit the MCMC sampling procedure to our problem setting 20, it is necessary to transform the problem of finding the global maximum from an initial point x to the problem of finding the global maximum within a range of $[b, e]$. One way to achieve this is to reject samples outside the range when sampling. Another approach, which we have adopted for its ease of implementation, is to introduce a transformer Ψ that maps $\mathbb{R}$ to [b,e]. In this way, if we find the minimal point of $B \circ \Psi$ at x^*, we have found the minimal point of B at $\Psi(x^*)$. Using such a transformer mapping allows us to apply a simple function transformation before feeding the energy function to an MCMC procedure, rather than modifying the MCMC internal, *i.e.*, the aforementioned second phase of the sampling. The rationale of this transformation can be summarized in the following lemma.

[6] Namely, ergodicity, detailed-balance, and aperiodicity [6].

Lemma 3. *Given f defined over an interval $[b,e]$, and the transformer Ψ defined over real numbers*

$$\Psi_{b,e}(x) \triangleq (b+e+(e-b)*\sin(x))/2. \tag{23}$$

If f has a global minimal point $x^* \in [b,e]$, we have

$$x^* = \Psi_{b,e}(\operatorname*{argmin}_{x\in\mathbb{R}}(f \circ \Psi_{b,e})). \tag{24}$$

As an example, if we want to find the minimal point of x^2 for $x \in [-1,1]$, we apply the transformer $\Psi_{-1,1}$, which is $\lambda x.\sin(x)$, and reduce the original optimization problem to finding the minimal point of $\sin^2(x)$ over $x \in \mathbb{R}$. This transformed problem is unconstrained as opposed to the original problem which limits x within a range. After finding the minimal point for the transformed problem, say, at $x^* = \pi$, we have also obtained the minimal point of the original problem, *i.e.*, $\Psi_{-1,1}(x^*) = 0$.

Algo. 2 shows our global BEA algorithm. We note that MCMC procedure itself never knows whether a real global optimum is found or not. Remind that the MCMC sampling only follows the desired distribution. If the energy function is expressed in the form of $\exp^{-\mathscr{E}}$, the points of lower energy will be sampled more frequently than the points of higher energy. Practically, we use more than one starting point to find the best global optimization. Lines 12-20 describe the process. The number of starting points is *n_start*. For each starting point, we generate a sequence of MCMC samples to find the maximum. The returned value of Algo. 2 is the largest obtained from all starting points.

Algorithm 2: Global BEA

	b	Lower bound of the searched interval
	e	Higher bound of the searched interval ($e \geq b$)
Input:	*n_start*	Number of starting points
	iter_global	Iteration bound for the procedure of `Metropolis-Hasting`
	B	Backward error computed in Algo. 1

Output: An MCMC estimation of $\max_{[b,e]} B$

1 Let $\Psi = \lambda x.(b+e+(e-b)*\sin(x))/2$
2 Let the energy function be

$$\mathscr{E} = \lambda x.\exp^{-B(\Psi(x)/T)}$$

/* Find the optimal MCMC solution for a grid of evenly spaced starting points */
3 **for** $j = 0$ *to* *n_start* -1 **do**
4 Let $x = \Psi^{-1}_{b,e}(b+(e-b)/(n\_start-1)*j)$
5 Let $x_{max}[j] = x$
/* Loop Invariant: $\mathscr{E}(x_{max}[j]) \geq \mathscr{E}(x)$ */
6 **for** $i = 1$ *to* *iter_global* **do**
/* Generate the next MCMC sample */
7 Let $x =$ `Metropolis-Hasting`$(\mathscr{E}, x)$
8 **if** $\mathscr{E}(x) > \mathscr{E}(x_{max}[j])$ **then**
9 $x_{max}[j] = x$
10 Let m be an index of the largest $x_{max}[\cdot]$, namely, $x_{max}[m] \geq x_{max}[m']$ for all $m' \in [0, n\_start-1]$
11 **return** $\Psi_{b,e}(x_{max}[m])$
12 **Procedure** `Metropolis-Hasting`$(\mathscr{E}, x)$
13 Let d be a random perturbation generated from a distribution predefined in the MCMC procedure
14 **if** $\mathscr{E}(x) < \mathscr{E}(x+d)$ **then**
15 Let *accept* $=$ *true*
16 **else**
17 Let r be a number generated from a uniform distribution $\mathscr{U}(0,1)$
18 Let *accept* be the Boolean

$$r < \frac{\mathscr{E}(x+d)}{\mathscr{E}(x)}$$

19 **if** *accept* **then return** $x+d$
20 **else return** x

4. Implementation and Evaluation

This section describes details of our implementation and presents our empirical evaluation results.

4.1 Implementation

We have implemented our backward error analysis using C++ and Python. Our implementation follows the high-level structure illustrated in Fig. 2, with the following components:

(C1) A *source transformer* that lifts the numerical code $\hat{f}$, of type `double`, to f, of higher precision, to simulate the exact solution. We have prototyped a source to source transformer that substitutes `double` with a higher precision type. For example, we can lift `y1` in Sect. 2 by changing its `double` to a higher-precision type, say `hp`

```
hp y1bis (hp x) {return sqrt(x + 1) - 1;}
```

Here, we have used the Argument-Dependent Lookup (ADL) feature of C++, assuming that the basic arithmetic operations and built-in functions like `sqrt` can be overloaded to operate on the high precision type `hp`. In our implementation, `hp` can be `cpp_dec_float <n>` ($n \in \mathbb{Z}$) of the Boost (v1.56.0) Multiprecision library [1], or the built-in `long double`.

(C2) A *backend of local BEA* which, given $\hat{f}$, f and x, finds the smallest $|\delta|$ so that $f(x+\delta\cdot x) \simeq f(x)$. As explained in Sect. 3, this step involves the reuse of a local minimizer `LM` and a root-finder `RF`. We use Brent's algorithm [12] for `LM`, and the bisection algorithm [36] for `RF`. Both of them are available from Boost.

(C3) A *forward error simulator* that simulates the forward error as $F(x) = |(f(x)-\hat{f}(x))/f(x)|$ given f, $\hat{f}$, and x. Note that $\hat{f}$ is typed differently from f, implying an implicit type conversion when computing F.

(C4) An *MCMC engine* that computes $\max_{x\in S}\mathscr{E}(x)$, where $\mathscr{E}(x)$ is a black-box energy function derived from either backward error $B(x)$ or condition number $C(x)$ (Eq. 7), and S refers to the search scope. We use the Basinhopping algorithm [42] implemented in Scipy (v0.15.0) [5] as the MCMC engine.

4.2 Empirical Results and Analysis

To demonstrate the effectiveness of our tool, we evaluate it on classic floating-point functions, including the transcendental functions of GNU C Library (glibc): `sin`, `cos`, `log`, `exp`, `sqrt` and two special functions `tgamma` and `erf` of the Boost library. The experiments were performed on a laptop with a 2.6 GHz Intel Core i7 and 16GB RAM running MacOS 10.9.5. The running time in our evaluation is measured using the C++ chrono library.

We find that the high precision type `hp` (Sect. 4.1) used in the source transformation step is crucial for the accuracy and efficiency of our assessment. Our evaluation studies three variants of our tool, BEA1000, BEA100 and BEA_L, that correspond to BEA with `hp` being `cpp_dec_float <1000>`, `cpp_dec_float <100>`, and `long double`.

In Sect. A, we give the exact values of the major parameters we have used in our experiments.

4.2.1 Assessment of Local BEA

Our goal of the first experiment is to find practical parameter settings for local BEA. Recall that local BEA yields $B(x)$ (Fig. 2), and we use it as a black-box function for global BEA and estimating the condition number. The performance of the last two crucially depends on that of local BEA. The setting of local BEA must, therefore, strike a balance between precision and efficiency.

To measure the accuracy of local BEA is tricky, because there have been no reported reference values of backward error for our tested functions. A workaround is to rely on the few classic functions, whose analytical form of condition numbers have been studied and known. See Tab. 4 for these condition numbers. Combined with a forward error F which we can simulate with Eq. (1), we use F/C as the theoretical value of backward error.

Tab. 5 reports the evaluation of local BEA. Each function under test (Col. 1) is analyzed with seven input points, 100, 10, 1, 0.1, 0.01, 0.001, and 0.0001 (Col. 2). As explained above, our evaluation adopts F/C as theoretical backward error (Col. 3-5). For each BEA1000, BEA100, and BEA_L, we give the local BEA results (Col. 6-8) and their running times (Col. 9-11). Some table entries are marked "N/A", *i.e.* "Not Applicable"[7].

We observe that, for transcendental functions of glbic, BEA1000 computes exactly the same backward error values as what are theoretically expected, namely, F/C. For this reason, we choose BEA1000 as reference in our subsequent comparison. Note that for special functions of Boost, F/C is unavailable, in which case we use BEA1000 as the reference.

[7] For example, we put N/A for sqrt (x) at x = 100 because its forward error returns 0 (*i.e.*, $\sqrt{100}$ is to be exactly computed), and therefore, local BEA returns 0 immediately (Algo. 1, Line 2); For tgamma or erf, no condition number is available to us (Tab. 4), hence we put N/A in Col. 3-5 of these functions in Tab. 5. Our metrics A_i and S_i ($i \in \{1,2\}$) are meaningless for these situations.

Table 4: Functions under test and their condition numbers in analytical forms [25].

Function		Condition number
glibc 2.21	cos	$x\sin(x)/\cos(x)$
	sin	$x\cos(x)/\sin(x)$
	exp	x
	log	$1/\log(x)$
	sqrt	0.5
Boost 1.56	tgamma	N/A
	erf	N/A

It can be seen that BEA100 reports almost the same backward error as BEA1000, meaning that BEA1000 is probably excessively precise. In terms of performance, BEA100 consumes only 1E-3 time as BEA1000 does (Col. 9). Again, this confirms that BEA1000 has only theoretical value (as a "very precise" backward analyzer), but far less practical than BEA100.

If we further relax the precision of the analyzer to `long double`, we find that the computed backward errors somewhat deviate from F/C, but are still of the same magnitude. The precision loss using BEA_L, however, is largely compensated by the performance gain. Comparing Col. 11 and 9, the time spent by BEA_L is almost 1E-6 times less. The gap between BEA_L and BEA1000 / BEA100 in terms of execution time is probably due to the fact that `long double` is implemented in hardware, while `cpp_dec_float` used in BEA100 or BEA1000 is multiprecision software that comes with a high performance penalty.

The metrics on the last four columns quantify the accuracy and performance gain. The parameters A_1, A_2 (Col. 7, 8) are the ratio between the backward error of BEA100 (resp. BEA_L) and BEA1000

$$A_1 \triangleq B_{\text{BEA100}}/B_{\text{BEA1000}} \qquad A_2 \triangleq B_{\text{BEA\_L}}/B_{\text{BEA1000}} \quad (25)$$

The parameter S_1 (resp. S_2) quantifies the speed-up:

$$S_1 \triangleq T_{\text{BEA100}}/T_{\text{BEA1000}}, \qquad S_2 \triangleq T_{\text{BEA\_L}}/T_{\text{BEA1000}} \quad (26)$$

The major conclusion we draw from this experiment is about how to choose the high precision type when performing backward error analysis. For global BEA of the next subsection, we will use `long double` due to its significant performance gain from hardware. For the subsection of condition number estimation, we will use BEA100 for its balance between precision and performance.

4.2.2 Assessment of Global BEA

This experiment evaluates the process of estimating maximum backward error for given intervals (Eq. 20). Because it is generally impossible to verify whether the global optimum has been reached unless the entire search scope is explored — impossible because simply there are too many floating-point

Table 5: Local BEA evaluation results for BEA1000, BEA100 and BEA_L. Theoretical condition number C is computed from the analytical form in Tab. 4, F is computed using Eq. (1). Accuracy Metrics include A_1, A_2 defined in Eq. 25. Performance Metrics include S_1, S_2 defined in Eq. 26.

		Function Characteristics			Backward Error (B)			Time (T, in second)			Metrics			
$\hat{f}$	x	C	F	F/C	B_{BEA1000}	B_{BEA100}	$B_{\text{BEA\_L}}$	T_{BEA1000}	T_{BEA100}	$T_{\text{BEA\_L}}$	A_1	A_2	S_1	S_2
cos(x)	100	5.87E+01	7.85E-17	1.34E-18	1.34E-18	1.34E-18	1.31E-18	2.53E+01	2.77E-02	1.90E-05	100.0%	98.4%	1.09E-03	7.51E-07
	10	6.48E+00	1.69E-17	2.60E-18	2.60E-18	2.60E-18	2.59E-18	2.48E+01	2.59E-02	1.00E-05	100.0%	99.6%	1.04E-03	4.03E-07
	1	1.56E+00	8.81E-17	5.66E-17	5.66E-17	5.66E-17	5.66E-17	2.61E+01	2.65E-02	8.00E-06	100.0%	100.1%	1.02E-03	3.07E-07
	0.1	1.00E-02	5.63E-17	5.61E-15	5.61E-15	5.61E-15	5.61E-15	2.39E+01	2.67E-02	9.00E-06	100.0%	100.0%	1.12E-03	3.77E-07
	0.01	1.00E-04	1.44E-17	1.44E-13	1.44E-13	1.44E-13	1.38E-13	3.35E+01	2.73E-02	8.00E-06	100.0%	95.7%	8.15E-04	2.39E-07
	0.001	1.00E-06	7.83E-18	7.83E-12	7.83E-12	7.83E-12	7.77E-12	3.24E+01	2.53E-02	8.00E-06	100.0%	99.2%	7.81E-04	2.47E-07
	0.0001	1.00E-08	2.62E-17	2.62E-09	2.62E-09	2.62E-09	2.62E-09	2.71E+01	2.23E-02	7.00E-06	100.0%	100.0%	8.23E-04	2.58E-07
sin(x)	100	1.70E+02	6.03E-18	3.54E-20	3.54E-20	3.54E-20	1.80E-20	2.28E+01	1.65E-02	1.60E-05	100.0%	50.9%	7.24E-04	7.02E-07
	10	1.54E+01	1.32E-16	8.59E-18	8.59E-18	8.59E-18	8.64E-18	2.34E+01	1.80E-02	9.00E-06	100.0%	100.5%	7.69E-04	3.85E-07
	1	6.42E-01	2.11E-18	3.29E-18	3.29E-18	3.29E-18	3.28E-18	2.71E+01	1.95E-02	1.90E-05	100.0%	99.9%	7.20E-04	7.01E-07
	0.1	9.97E-01	3.09E-17	3.10E-17	3.10E-17	3.10E-17	3.10E-17	4.97E+01	3.02E-02	7.00E-06	100.0%	100.1%	6.08E-04	1.41E-07
	0.01	1.00E+00	4.27E-17	4.27E-17	4.27E-17	4.27E-17	4.28E-17	3.89E+01	2.33E-02	6.00E-06	100.0%	100.2%	5.99E-04	1.54E-07
	0.001	1.00E+00	5.67E-18	5.67E-18	5.67E-18	5.67E-18	5.62E-18	3.14E+01	1.86E-02	6.00E-06	100.0%	99.1%	5.92E-04	1.91E-07
	0.0001	1.00E+00	5.10E-17	5.10E-17	5.10E-17	5.10E-17	5.10E-17	2.41E+01	1.38E-02	6.00E-06	100.0%	100.0%	5.73E-04	2.49E-07
exp(x)	100	1.00E+02	5.99E-17	5.99E-19	5.99E-19	5.99E-19	6.37E-19	9.59E+00	1.25E-02	1.80E-05	100.0%	106.3%	1.30E-03	1.88E-06
	10	1.00E+01	6.26E-17	6.26E-18	6.26E-18	6.26E-18	6.25E-18	9.38E+00	8.81E-03	9.00E-06	100.0%	99.9%	9.39E-04	9.59E-07
	1	1.00E+00	5.32E-17	5.32E-17	5.32E-17	5.32E-17	5.32E-17	2.07E+01	1.61E-02	9.00E-06	100.0%	99.9%	7.78E-04	4.35E-07
	0.1	1.00E-01	7.37E-17	7.37E-16	7.37E-16	7.37E-16	7.37E-16	1.38E+01	1.13E-02	9.00E-06	100.0%	99.9%	8.19E-04	6.52E-07
	0.01	1.00E-02	1.08E-16	1.08E-14	1.08E-14	1.08E-14	1.08E-14	1.21E+01	8.69E-03	8.00E-06	100.0%	100.0%	7.18E-04	6.61E-07
	0.001	1.00E-03	4.29E-17	4.29E-14	4.29E-14	4.29E-14	2.28E-14	8.88E+00	6.54E-03	8.00E-06	100.0%	53.2%	7.36E-04	9.01E-07
	0.0001	1.00E-04	4.33E-17	4.33E-13	4.33E-13	4.33E-13	4.15E-13	7.55E+00	6.22E-03	8.00E-06	100.0%	95.9%	8.24E-04	1.06E-06
log(x)	100	2.17E-01	9.43E-17	4.34E-16	4.34E-16	4.34E-16	4.34E-16	3.21E+02	1.22E-01	2.00E-05	100.0%	100.0%	3.80E-04	6.23E-08
	10	4.34E-01	9.43E-17	2.17E-16	2.17E-16	2.17E-16	2.17E-16	3.49E+02	1.34E-01	1.20E-05	100.0%	100.0%	3.84E-04	3.44E-08
	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	0.1	4.34E-01	7.45E-17	1.72E-16	1.72E-16	1.72E-16	1.71E-16	2.99E+02	1.12E-01	1.10E-05	100.0%	100.0%	3.75E-04	3.68E-08
	0.01	2.17E-01	9.41E-17	4.33E-16	4.33E-16	4.33E-16	4.33E-16	3.73E+02	1.37E-01	1.10E-05	100.0%	99.9%	3.67E-04	2.95E-08
	0.001	1.45E-01	3.13E-17	2.16E-16	2.16E-16	2.16E-16	2.17E-16	1.34E+02	5.19E-02	1.10E-05	100.0%	100.4%	3.87E-04	8.21E-08
	0.0001	1.09E-01	9.34E-17	8.60E-16	8.60E-16	8.60E-16	8.60E-16	2.85E+02	1.06E-01	1.10E-05	100.0%	100.0%	3.72E-04	3.86E-08
sqrt(x)	100	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	10	5.00E-01	6.03E-17	1.21E-16	1.21E-16	1.21E-16	1.21E-16	1.42E+00	6.35E-03	1.60E-05	100.0%	100.2%	4.47E-03	1.13E-05
	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	0.1	5.00E-01	2.53E-18	5.06E-18	5.06E-18	5.06E-18	5.06E-18	1.64E+00	7.46E-03	8.00E-06	100.0%	100.0%	4.55E-03	4.88E-06
	0.01	5.00E-01	4.51E-17	9.02E-17	9.02E-17	9.02E-17	9.02E-17	1.42E+00	6.97E-03	7.00E-06	100.0%	100.0%	4.91E-03	4.93E-06
	0.001	5.00E-01	7.30E-17	1.46E-16	1.46E-16	1.46E-16	1.46E-16	1.62E+00	7.78E-03	7.00E-06	100.0%	100.1%	4.80E-03	4.32E-06
	0.0001	5.00E-01	3.14E-18	6.29E-18	6.29E-18	6.29E-18	6.28E-18	1.59E+00	7.34E-03	7.00E-06	100.0%	99.8%	4.62E-03	4.40E-06
tgamma	100	N/A	N/A	N/A	2.65E-18	2.65E-18	2.68E-18	3.30E+01	1.60E-01	2.10E-05	100.00%	101.28%	4.85E-03	6.36E-07
	10	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	0.1	N/A	N/A	N/A	7.08E-17	7.08E-17	7.08E-17	3.35E+01	2.16E-01	1.00E-05	100.00%	99.90%	6.45E-03	2.99E-07
	0.01	N/A	N/A	N/A	1.95E-18	1.95E-18	1.97E-18	3.35E+01	1.73E-01	1.70E-05	100.00%	101.33%	5.16E-03	5.07E-07
	0.001	N/A	N/A	N/A	2.92E-17	2.92E-17	2.91E-17	3.35E+01	1.78E-01	9.00E-06	100.00%	99.81%	5.31E-03	2.69E-07
	0.0001	N/A	N/A	N/A	2.16E-17	2.16E-17	2.16E-17	3.35E+01	1.75E-01	9.00E-06	100.00%	100.02%	5.22E-03	2.69E-07
erf	100	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	10	N/A	N/A	N/A	1.00E-04	1.00E-04	0.00E+00	4.15E+01	3.86E-02	1.00E-06	100.00%	0.00%	9.30E-04	2.41E-08
	1	N/A	N/A	N/A	2.08E-16	2.08E-16	2.08E-16	3.88E+01	2.08E-01	2.90E-05	100.00%	100.17%	5.36E-03	7.47E-07
	0.1	N/A	N/A	N/A	8.28E-18	8.28E-18	8.38E-18	2.76E+00	1.82E-02	1.10E-05	100.00%	101.13%	6.59E-03	3.99E-06
	0.01	N/A	N/A	N/A	9.32E-17	9.32E-17	9.31E-17	1.88E+00	1.24E-02	1.00E-05	100.00%	99.94%	6.60E-03	5.32E-06
	0.001	N/A	N/A	N/A	6.99E-17	6.99E-17	6.98E-17	1.43E+00	9.84E-03	1.10E-05	100.00%	99.86%	6.88E-03	7.69E-06
	0.0001	N/A	N/A	N/A	2.54E-16	2.54E-16	2.54E-16	1.17E+00	8.46E-03	1.00E-05	100.00%	99.98%	7.23E-03	8.55E-06

numbers — we content ourselves with the following consistency check: If we have a decreasing sequence of intervals $I1 \supseteq I2 \ldots$, the computed maximal backward error must be non-increasing

$$I_1 \supseteq I_2 \implies \max_{x \in I_1} B(x) \geq \max_{x \in I_2} B(x) \tag{27}$$

We evaluate global BEA on the same functions as in local BEA, *i.e.*, `sin`, `cos`, `exp`, `log`, `sqrt`, `tgamma` and `erf`. For each, we estimate $\max_{x \in I} B(x)$ for I being [0.0001,100], [0.0001,10], [0.0001, 1], [0.0001, 0.1], [0.0001, 0.01] and [0.0001,0.001]. Usually, the number of sampling is crucial for MCMC approaches. The sampling number of our global BEA is controlled by the parameter iter_global (Algo. 2). We evaluate the global BEA with iter_global = 100, 1000 and 10000. As mentioned earlier, we will use BEA_L in the evaluation.

We observe that the results of our global BEA are overall consistent. When iter_global = 1000 or 10000, the results are consistent for all the tested functions. When iter_global = 100, for iter_global = 100, we have 6 out of 7 tested functions that exhibit consistent results. The only exception is for `exp` (Col. 4) for the interval [0.0001,1], when iter_global = 100. The maximum backward error for `exp` (Col. 4) is found to be 9.64E-13 for the interval [0.0001,1], but then $B(x^*)$ increases to 1.00E-12 when the search interval is [0.0001, 0.1]. The issue is due to the insufficient number of iterations. Note that the inconsistency disappears when the iteration number increases to 1000 (Col. 7).

The running time of global BEA is less than 2 seconds for iter_global = 100, less than 20 seconds when iter_global = 1000 (19.5s for `tgamma` of [0.0001,100]). The worst-case running time is attained when iter_global = 10000, for `tgamma` estimated over [0.0001, 1] (133 seconds).

In conclusion, for most tested functions and the decreasing sequence of intervals, the magnitude of $B(x^*)$ decreases or remains the same, as expected. Because strong consistency has already been obtained when iter_global = 1000, we use iter_global = 1000 in practice.

4.2.3 Estimating the Condition Number

Sanity Check. As explained in Sect. 1, the condition number can be computed as $C(x) = F(x)/B(x)$. An appealing feature of condition number is that it does not depend on any particular implementation, but the problem itself.

As a sanity check, we verify that using the two implementations of $\sin(x)$ discussed in Sect. 2, namely, glibc's `sin` and `fsin` instruction of Intel x87, we produce the same condition number. Let $F_1(x)$ and $B_1(x)$ denote the forward error and backward error of Intel `fsin`, and $F_2(x)$ and $B_2(x)$ denote those of glibc `sin`. Then, the sanity check consists in verifying

$$F_1(x)/B_1(x) \simeq F_2(x)/B_2(x) \tag{28}$$

for all input x. For most input x, our results show that $F_1(x)$ is very close to $F_2(x)$ and $B_1(x)$ is very close to $B_2(x)$, meaning that (28) trivially holds for those input. Nontrivial cases can be observed when x is close to π, which we present in Tab. 7. For a sequence of input points closer and closer to π, we compute the condition numbers of `sin` and `fsin` (Col. 2-4 and Col. 5-7 respectively). Observe that $F_1(x) = F_2(x)$ when $x = 3, 3.14, 3.141$ and 3.1415, and the same holds for $B_1(x)$ and $B_2(x)$; then, after $x = 3.14159$, $F_1(x)$ shrinks under machine epsilon, whereas $B_1(x)$ changes very little. On the other hand, $B_2(x)$ decreases rather quickly after $x = 3.14159$, while $F_2(x)$ remains stable. What remains invariant is that $F_1(x)/B_1(x)$ and $F_2(x)/B_2(x)$ remain almost equal throughout (compare Col. 4 and 7). We quantify their difference by reporting C_1/C_2 in the last column, where $C_1 = F_1/B_1$ and $C_2 = F_2/B_2$. It can be seen that C_1/C_2 is almost 1 for all input points. This confirms Eq. (28).

Global Condition Estimation. If we feed $C(x)$ into a MCMC procedure, as what we have done for global BEA, we can estimate the maximal condition number within an interval. We call this process *global condition estimation*. Such an example is shown in Sect. 2, Tab. 2. Next we will study experimental results of global condition estimation.

Unlike global BEA that uses BEA_L which achieves a high efficiency, we have to settle for the less efficient, but more precise BEA100 for estimating the condition number over a given interval. The reason is that BEA_L is unable to detect backward error smaller than the machine epsilon of `long double`, *i.e.*, 10^{-19}. Failing to estimate very small backward errors does not cause an issue for global BEA which aims at finding maximal backward error, but is problematic for estimating the condition number, F/B (where B is the denominator).

Tab. 8 shows the experimental results. We have used the same set of functions and intervals as in the global BEA experiment (Col. 1, 2). With BEA100, we only evaluate our analysis for iter_global = 100 (Col. 3-5) and 10000 (Col. 6-8).

Let us take a close look at the maximum condition numbers of the transcendental functions. Using their analytical forms in Tab. 4, we can verify the correctness of our results. The analytical form of condition number for $\cos(x)$ is $x\tan(x)$, which tends to infinity when x is close to $(0.5 + k) * \pi$. We have obtained, for the interval [0.0001,100], x*=7.07E+01,*i.e.*, 22.50π. For the interval [0.001,10], we obtain x*=4.71, which is $1.50*\pi$. For intervals [0.0001, 1], [0.0001,0.1], [0.0001,0.01] and [0.0001,001], we obtain the largest condition number at 0.001 for all these ranges. This is as expected, corresponding to the fact that $x\tan(x)$ monotonically decreases when $x \leq 1$. The $\sin(x)$ function has the analytical condition number $x/\tan(x)$. Hence, sin is ill-conditioned, and its condition number peaks at $x = k\pi$, and is increasingly monotone for $x \leq 1$. We have computed the maximal point of the condition number being 9.74E+01= $31.00*\pi$, and 9.42=3.00 $*\pi$ for the ranges [0.0001,100] and [0.001,10], respectively. The maximal point reaches the right border of the ranges that are subsets of [0.0001,1], as ex-

Table 6: Global BEA assessment. x*: point where the maximal backward error is reached, and time used for the estimation T (in seconds).

		iter_global =100			iter_global = 1000			iter_global = 10000		
$\hat{f}$	$[b,e]$	x^*	$B(x^*)$	T (s)	x^*	$B(x^*)$	T (second)	x^*	$B(x^*)$	T (s)
cos	[0.0001,100]	5.65E+01	1.00E-04	9.45E-01	8.80E+01	1.00E-04	2.82E-01	7.85E+01	1.00E-04	1.37E+00
	[0.0001,10]	1.00E-04	5.04E-09	1.37E+00	9.42E+00	1.00E-04	3.34E+00	9.42E+00	1.00E-04	4.10E+00
	[0.0001,1]	1.01E-04	5.21E-09	1.32E+00	1.01E-04	5.46E-09	1.35E+01	1.00E-04	5.53E-09	1.36E+02
	[0.0001,0.1]	1.00E-04	5.40E-09	1.38E+00	1.00E-04	5.46E-09	1.36E+01	1.00E-04	5.51E-09	1.32E+02
	[0.0001,0.01]	1.01E-04	5.43E-09	1.31E+00	1.00E-04	5.52E-09	1.35E+01	1.00E-04	5.55E-09	1.31E+02
	[0.0001,0.001]	1.00E-04	5.49E-09	1.34E+00	1.00E-04	5.54E-09	1.35E+01	1.00E-04	5.55E-09	1.33E+02
sin	[0.0001,100]	8.33E+01	1.00E-04	3.52E-01	4.87E+01	1.00E-04	2.85E-01	9.58E+01	1.00E-04	1.52E+00
	[0.0001,10]	1.57E+00	1.95E-13	1.35E+00	7.85E+00	1.00E-04	5.04E+00	7.85E+00	1.00E-04	1.41E+00
	[0.0001,1]	8.33E-01	2.62E-16	1.28E+00	8.01E-01	2.76E-16	1.37E+01	7.88E-01	2.81E-16	1.37E+02
	[0.0001,0.1]	6.28E-02	1.08E-16	1.19E+00	6.27E-02	1.11E-16	1.28E+01	6.26E-02	1.12E-16	1.28E+02
	[0.0001,0.01]	7.90E-03	1.10E-16	1.28E+00	3.92E-03	1.10E-16	1.30E+01	7.82E-03	1.11E-16	1.30E+02
	[0.0001,0.001]	1.24E-04	1.09E-16	1.25E+00	9.77E-04	1.11E-16	1.29E+01	9.77E-04	1.11E-16	1.26E+02
exp	[0.0001,100]	1.02E-04	8.61E-13	1.32E+00	1.00E-04	1.05E-12	1.33E+01	1.00E-04	1.10E-12	1.38E+02
	[0.0001,10]	1.06E-04	8.51E-13	1.28E+00	1.00E-04	1.07E-12	1.34E+01	1.01E-04	1.10E-12	1.36E+02
	[0.0001,1]	1.00E-04	9.63E-13	1.42E+00	1.00E-04	1.07E-12	1.37E+01	1.01E-04	1.10E-12	1.38E+02
	[0.0001,0.1]	1.07E-04	1.00E-12	1.37E+00	1.01E-04	1.09E-12	1.38E+01	1.00E-04	1.11E-12	1.41E+02
	[0.0001,0.01]	1.00E-04	1.10E-12	1.35E+00	1.00E-04	1.10E-12	1.33E+01	1.00E-04	1.11E-12	1.37E+02
	[0.0001,0.001]	1.01E-04	1.10E-12	1.40E+00	1.00E-04	1.11E-12	1.32E+01	1.00E-04	1.11E-12	1.36E+02
log	[0.0001,100]	1.00E-04	8.66E-16	1.43E+00	2.81E-04	8.80E-16	1.37E+01	3.18E-04	8.86E-16	1.34E+02
	[0.0001,10]	2.44E-04	8.68E-16	1.49E+00	2.27E-04	8.87E-16	1.35E+01	2.48E-04	8.87E-16	1.34E+02
	[0.0001,1]	1.78E-04	8.88E-16	1.47E+00	3.05E-04	8.88E-16	1.38E+01	3.03E-04	8.88E-16	1.34E+02
	[0.0001,0.1]	1.20E-04	8.79E-16	1.45E+00	2.26E-04	8.87E-16	1.37E+01	1.11E-04	8.88E-16	1.33E+02
	[0.0001,0.01]	3.16E-04	8.85E-16	1.51E+00	1.07E-04	8.88E-16	1.35E+01	2.76E-04	8.88E-16	1.32E+02
	[0.0001,0.001]	1.00E-04	8.88E-16	1.43E+00	1.03E-04	8.88E-16	1.35E+01	1.01E-04	8.88E-16	1.32E+02
sqrt	[0.0001,100]	6.41E+01	2.21E-16	1.31E+00	6.41E+01	2.21E-16	1.14E+01	4.01E+00	2.22E-16	1.18E+02
	[0.0001,10]	4.18E+00	2.15E-16	1.20E+00	4.02E+00	2.21E-16	1.17E+01	4.01E+00	2.22E-16	1.15E+02
	[0.0001,1]	6.35E-02	2.19E-16	1.23E+00	2.51E-01	2.21E-16	1.12E+01	2.50E-01	2.22E-16	1.17E+02
	[0.0001,0.1]	1.60E-02	2.18E-16	1.17E+00	6.26E-02	2.21E-16	1.13E+01	1.56E-02	2.22E-16	1.17E+02
	[0.0001,0.01]	3.93E-03	2.20E-16	1.16E+00	9.88E-04	2.20E-16	1.12E+01	9.79E-04	2.21E-16	1.20E+02
	[0.0001,0.001]	9.78E-04	2.21E-16	1.23E+00	9.78E-04	2.22E-16	1.14E+01	9.77E-04	2.22E-16	1.18E+02
tgamma	[0.0001,100]	1.46E+00	1.43E-05	2.07E+00	1.46E+00	1.61E-12	1.95E+01	1.46E+00	1.00E-04	1.18E+02
	[0.0001,10]	1.46E+00	7.05E-14	1.47E+00	1.46E+00	5.94E-13	1.37E+01	1.46E+00	1.00E-04	6.93E+01
	[0.0001,1]	9.98E-01	3.79E-16	1.42E+00	9.91E-01	4.11E-16	1.32E+01	1.00E+00	4.38E-16	1.33E+02
	[0.0001,0.1]	5.99E-02	1.56E-16	1.41E+00	5.84E-02	1.60E-16	1.33E+01	1.55E-02	1.66E-16	1.32E+02
	[0.0001,0.01]	7.56E-03	1.56E-16	1.37E+00	7.72E-03	1.64E-16	1.32E+01	1.21E-04	1.63E-16	1.29E+02
	[0.0001,0.001]	9.73E-04	1.62E-16	1.37E+00	9.57E-04	1.62E-16	1.33E+01	9.75E-04	1.65E-16	1.31E+02
erf	[0.0001,100]	6.41E+00	1.00E-04	2.30E-02	6.43E+00	1.00E-04	1.12E-01	6.34E+00	1.00E-04	1.09E+00
	[0.0001,10]	6.41E+00	1.00E-04	1.47E-02	6.38E+00	1.00E-04	1.43E-01	6.16E+00	1.00E-04	1.40E+00
	[0.0001,1]	4.76E-01	5.10E-16	1.55E+00	4.78E-01	6.05E-16	1.55E+01	4.73E-01	6.20E-16	1.53E+02
	[0.0001,0.1]	2.91E-02	3.81E-16	1.24E+00	5.08E-03	4.04E-16	1.23E+01	3.18E-04	4.17E-16	1.22E+02
	[0.0001,0.01]	1.40E-03	3.90E-16	1.24E+00	9.97E-03	3.94E-16	1.20E+01	1.00E-02	4.24E-16	1.19E+02
	[0.0001,0.001]	1.62E-04	3.95E-16	1.30E+00	1.10E-04	4.00E-16	1.20E+01	4.40E-04	4.22E-16	1.21E+02

pected. The condition number for $\log(x)$ is $1/\log(x)$, which has a singularity at $x = 1$ near which it is unbounded. Our analysis captures the largest condition number for the search interval [0.0001,1] and higher. For the other intervals, the largest condition numbers are obtained at the right boundary of the searched intervals, which is also as expected since $1/log(x)$ monotonically decreases when $x < 1$. The condition number for $\exp(x)$ is simply x. Our analyzer returns e for the searched interval $[b,e]$. The analytical condition number for $\text{sqrt}(x)$ is constant 0.5. The maximal point of the condition number can, therefore, be any.

To sum up, our analysis yields a tight estimation of $\max_{x\in I} C(x)$. The running time for each estimation of glibc's functions are within minutes. More time is needed for estimating the special functions (about 4.5 hours on [0.0001,0.01] for `tgamma`). Since condition number estimation is notoriously a hard problem, the time spent on them should be justified by their benefits.

5. Related Work

This section surveys related research and further positions our work. Miller presents a BEA algorithm [33] for straight-line

Table 7: Comparing Intel `fsin` and glibc (v 2.21) `sin` for a sequence of inputs close to *pi*. The columns of F, B, and C refer to relative forward error, relative backward error and condition number, respectively.

	Intel `fsin`			glibc `sin`			
Input	$F1$	B_1	$C_1(F_1/B_1)$	F_2	B_2	$C_2(F_2/B_2)$	C_1/C_2
3	6.0779970E-17	2.8879915E-18	2.1045758E+01	6.0779970E-17	2.8879915E-18	2.1045758E+01	1.0000000E+00
3.1	1.7095339E-17	2.2950026E-19	7.4489409E+01	1.7095339E-17	2.2950026E-19	7.4489409E+01	1.0000000E+00
3.14	1.8632376E-17	9.4506195E-21	1.9715507E+03	1.8632376E-17	9.4506195E-21	1.9715507E+03	1.0000000E+00
3.141	3.3602131E-17	6.3401546E-21	5.2998913E+03	3.3602131E-17	6.3401546E-21	5.2998913E+03	1.0000000E+00
3.1415	8.5276377E-17	2.5150923E-21	3.3905864E+04	8.5276377E-17	2.5150923E-21	3.3905864E+04	1.0000000E+00
3.14159	1.4829918E-15	1.2526307E-21	1.1839019E+06	1.1302151E-16	9.5465265E-23	1.1839019E+06	1.0000000E+00
3.141592	6.1079480E-15	1.2707228E-21	4.8066724E+06	4.7910788E-17	9.9675584E-24	4.8066724E+06	1.0000000E+00
3.1415926	7.5487357E-14	1.2876755E-21	5.8622966E+07	3.9055811E-17	6.6622032E-25	5.8622966E+07	1.0000000E+00
3.14159265	1.1266735E-12	1.2874122E-21	8.7514590E+08	7.0150194E-18	8.0158293E-27	8.7514580E+08	1.0000001E+00
3.141592653	6.8575431E-12	1.2874147E-21	5.3266001E+09	5.5711593E-17	1.0459128E-26	5.3266001E+09	1.0000000E+00
3.1415926535	4.5042756E-11	1.2874147E-21	3.4986983E+10	6.5502722E-17	1.8722026E-27	3.4986984E+10	9.9999999E-01
3.14159265358	4.1299490E-10	1.2874147E-21	3.2079400E+11	5.9303635E-17	1.8486478E-28	3.2079466E+11	9.9999794E-01
3.141592653589	5.0985843E-09	1.2874147E-21	3.9603279E+12	3.1608549E-17	7.9813198E-30	3.9603160E+12	1.0000030E+00
3.1415926535897	4.3312066E-08	1.2874147E-21	3.3642669E+13	1.8158747E-18	5.3975347E-32	3.3642668E+13	1.0000000E+00
3.14159265358979	1.2517567E-06	1.2874147E-21	9.7230266E+14	5.2480308E-17	5.3921925E-32	9.7326474E+14	9.9901150E-01

programs that have no control flow. Gáti attempts to relieve this limit but in the price of high overhead. Gáti generates a straight-line program per a single program input, and then apply Miller's algorithm as a black-box [20]. In contrast, our approach comes with a BEA formulation with mathematical optimization, which, combined with MCMC sampling, deals with general floating-point programs.

Backward Error Analysis. Wilkinson's work on the foundation of backward error analysis has its root in *Error Analysis in Floating Point Arithmetic* [44], and his research on floating-point program error analysis, culminating in his influential paper *Rounding Errors in Algebraic Process* [43] and Turing Award in 1970. BEA has been continued [18, 29] and becomes a Swiss army knife for dealing with many different types of uncertainty computations [28, 41]. The technical details of BEA are summarized in [21], [26], [35].

The idea of *automated* error analysis goes back to the dawn of scientific computing, for example, see [46] for a running error analysis technique where an error bound is computed concurrently with the solution. Over years, various techniques of automated error analysis have been developed. Most techniques target specific mathematical quantities that measure the accuracy or stability of numerical computation, such as direct search optimization techniques for studying the growth factor of Gaussian elimination and condition number of a matrix. In contrast, our approach presented in this paper targets generic numerical code. Our automated BEA benefits both developers and numerical analysts, and complements other program analysis approaches for floating-point programs.

Static Analysis. Static analysis of a floating-point program consists in automatically deriving the possible values of program variables during its execution [24, 34]. Such analyses allow the detection of a large class of bugs or for proving their absence [10], and form the basis of more sophisticated analyses [23] and program transformations [32]. The problem of finding the exact set of values is known to be undecidable, and approximate solutions have been extensively studied, especially in the framework of abstract interpretation [14, 15], which provides a mathematical foundation for reasoning about approximations and their computation.

Static approaches are attractive because of their soundness guarantees. Usually, however, such soundness information is too conservative to be useful. Another disadvantage is the limited language features supported by most static analyzers. For example, few static analyzers can precisely deal with programs with pointers. Note that this limitation can be theoretical [19]: classic static numerical analysis has to be extended with pointer-aware abstract domains, but the extended analyzers unavoidably lose precision, in particular, when handling numerical operations on the heap.

Compared with static approaches, our BEA technique requires program execution, produces quantitative answers, and has no theoretical limitation for the kind of programs under analysis.

Runtime Techniques. A number of dynamic or symbolic approaches exist for analyzing floating-point programs. We discuss a few recent, representative efforts. Barr *et al.* [8] use symbolic execution [16, 30] to discover program inputs that trigger runtime floating-point exceptions. Tang *et al.* [40] discover potential instability issues by systematically altering the intermediate values or expressions of numerical computation.

Benz *et al.* [9], in contrary, try to assess numerical accuracy by a side-by-side runtime monitoring of computational precision. Bao and Zhang [7] propose a technique to reduce the cost of such runtime detection by not explicitly computing the precise error, but rather marking and tracking potentially inaccurate values. Chiang *et al.* [13] develop a heuristic algorithm to find inputs that lead to large forward error.

Table 8: Estimate of maximal condition number within a search interval. The maximal point x^*, maximal condition number $C(x^*)$ and the consumed time T are measured in for iteration iter_global = 100 and iter_global = 1000.

		iter_global = 100			iter_global = 1000		
f	$[b,e]$	x^*	$C(x^*)$	T (s)	x^*	$C(x^*)$	T (s)
cos	[0.0001,100]	9.27E+01	1.06E+04	46.02	7.07E+01	1.43E+06	733.42
	[0.0001,10]	7.85E+00	1.92E+04	56.01	4.71E+00	4.31E+05	820.70
	[0.0001,1]	1.00E+00	1.56E+00	55.24	1.00E+00	1.56E+00	738.51
	[0.0001,0.1]	1.00E-01	1.00E-02	85.15	1.00E-01	1.00E-02	763.73
	[0.0001,0.01]	1.00E-02	1.00E-04	67.83	1.00E-02	1.00E-04	478.61
	[0.0001,0.001]	1.00E-03	1.00E-06	51.26	1.00E-03	1.00E-06	395.46
sin	[0.0001,100]	9.11E+01	1.43E+05	47.10	9.74E+01	3.38E+05	736.20
	[0.0001,10]	9.43E+00	1.26E+04	56.43	9.42E+00	1.71E+05	829.29
	[0.0001,1]	1.00E-04	1.00E+00	57.18	1.00E-04	1.00E+00	753.19
	[0.0001,0.1]	1.00E-04	1.00E+00	80.44	1.00E-04	1.00E+00	711.33
	[0.0001,0.01]	1.00E-04	1.00E+00	67.63	1.00E-04	1.00E+00	464.98
	[0.0001,0.001]	1.00E-04	1.00E+00	48.04	1.00E-04	1.00E+00	404.15
exp	[0.0001,100]	1.00E+02	1.00E+02	24.92	1.00E+02	1.00E+02	441.57
	[0.0001,10]	1.00E+01	1.00E+01	24.81	1.00E+01	1.00E+01	461.60
	[0.0001,1]	1.00E+00	1.00E+00	34.12	1.00E+00	1.00E+00	575.59
	[0.0001,0.1]	1.00E-01	1.00E-01	25.15	1.00E-01	1.00E-01	420.89
	[0.0001,0.01]	1.00E-02	1.00E-02	20.33	1.00E-02	1.00E-02	345.76
	[0.0001,0.001]	1.00E-03	1.00E-03	17.66	1.00E-03	1.00E-03	293.57
log	[0.0001,100]	1.01E+00	1.40E+02	205.43	9.99E-01	1.47E+03	3282.17
	[0.0001,10]	9.96E-01	2.22E+02	194.73	1.00E+00	6.49E+03	1962.48
	[0.0001,1]	1.00E+00	1.06E+07	145.93	1.00E+00	5.28E+09	748.40
	[0.0001,0.1]	1.00E-01	4.34E-01	294.70	1.00E-01	4.34E-01	3681.57
	[0.0001,0.01]	1.00E-02	2.17E-01	220.50	1.00E-02	2.17E-01	2201.92
	[0.0001,0.001]	1.00E-03	1.45E-01	177.97	1.00E-03	1.45E-01	1699.71
sqrt	[0.0001,100]	3.29E+01	5.00E-01	13.95	3.96E+01	5.00E-01	144.63
	[0.0001,10]	8.28E+00	5.00E-01	15.83	1.64E+00	5.00E-01	155.57
	[0.0001,1]	9.53E-01	5.00E-01	16.03	6.53E-01	5.00E-01	157.12
	[0.0001,0.1]	8.39E-02	5.00E-01	15.78	7.36E-04	5.00E-01	158.95
	[0.0001,0.01]	8.80E-03	5.00E-01	15.51	2.97E-03	5.00E-01	155.76
	[0.0001,0.001]	1.10E-04	5.00E-01	17.85	7.84E-04	5.00E-01	151.64
tgamma	[0.0001,100]	1.00E+02	4.60E+02	599.24	1.00E+02	4.60E+02	5235.82
	[0.0001,10]	1.00E+01	2.25E+01	1104.69	1.00E+01	2.25E+01	5559.69
	[0.0001,1]	2.16E-01	1.06E+00	1173.08	2.16E-01	1.06E+00	5631.08
	[0.0001,0.1]	1.00E-01	1.04E+00	824.38	1.00E-01	1.04E+00	5605.60
	[0.0001,0.01]	1.00E-02	1.01E+00	678.19	1.00E-02	1.01E+00	16060.74
	[0.0001,0.001]	1.00E-03	1.00E+00	609.94	1.00E-03	1.00E+00	5637.66
erf	[0.0001,100]	1.00E-04	1.00E+00	821.89	1.00E-04	1.00E+00	10663.07
	[0.0001,10]	1.00E-04	1.00E+00	1589.78	1.00E-04	1.00E+00	9685.16
	[0.0001,1]	1.00E-04	1.00E+00	537.75	1.00E-04	1.00E+00	2919.77
	[0.0001,0.1]	1.00E-04	1.00E+00	130.62	1.00E-04	1.00E+00	647.44
	[0.0001,0.01]	1.00E-04	1.00E+00	84.58	1.00E-04	1.00E+00	489.99
	[0.0001,0.001]	1.00E-04	1.00E+00	60.17	1.00E-04	1.00E+00	422.12

Rubio-Gonzalez *et al.* [37] aim to enhance performance of floating-point programs by tuning the types of floating-point variables. Schkufza *et al.* [39] propose a technique to automatically tune the precision of floating-point code for compiler optimization to allow an acceptable loss of precision. Zou *et al.* [48] use fitness functions and genetic algorithms to generate inputs of floating point programs. These inputs are then used to trigger inaccuracies in the programs.

While the exploration of run-time properties allows dynamic approaches to carry out a fine-grained analysis of floating-point programs, these approaches focus on forward error, *i.e.* the difference between the expected and the real output, which has been recognized by numerical analysts as less powerful than performing error analysis *à la backward*.

6. Conclusion

We have presented an automated backward error analysis (abbreviated as BEA in the paper) for analyzing floating-point programs. We have considered both *local* and *global* backward error analysis. The local analysis focuses on understanding detailed characteristics of a numerical program at a single point. It not only provides insight into program behavior for a single point and its neighborhood, but also supports the global analysis, *i.e.*, the estimation of backward error bounds across a whole input range.

As application, we have also studied condition number estimation, and applied it to some well-known inaccuracy issues of Intel FPU `fsin` instruction. Our experimental results validate the effectiveness of our approach and demonstrate its utility in understanding floating-point programs.

While the theory of this work is presented under the one-dimensional context, BEA as we have presented should be generally applicable to functions of higher dimensions. We plan to extend our analysis scope to $\mathbb{R}^n$ in order to handle functions that operate on vectors and matrices. In addition to this, we also plan to apply BEA to find and understand unknown issues in legacy numerical code.

Acknowledgments

We thank the anonymous reviewers for their useful comments on earlier versions of this paper. Our special thanks go to Hanfei Wang for his initial participation on this project and for his thoughtful feedback. We also gratefully acknowledge Mehrdad Afshari for his help in setting up our experiment evaluation with the GNU C Library (glibc).

This work was supported in part by NSF Grant No. 1349528. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] Boost multi-precision package. `http://www.boost.org/doc/libs/1_57_0/libs/multiprecision/doc/html/index.html`. Retrieved: 25 Mar 2015.

[2] Intel underestimates error bounds by 1.3 quintillion. `https://randomascii.wordpress.com/2014/10/09/intel-underestimates-error-bounds-by-1-3-quintillion/`. Retrieved: 25 Mar 2015.

[3] `https://software.intel.com/blogs/2014/10/09/fsin-documentation-improvements-in-the-intel-64-and-ia-32-architectures-software`. Retrieved: 25 Mar 2015.

[4] `https://sourceware.org/bugzilla/show_bug.cgi?id=13658`. Retrieved: 25 Mar 2015.

[5] Scipy optimization package. `http://docs.scipy.org/doc/scipy-dev/reference/optimize.html#module-scipy.optimize`. Retrieved: 25 Mar 2015.

[6] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50(1-2):5–43, 2003.

[7] T. Bao and X. Zhang. On-the-fly detection of instability problems in floating-point program execution. In *OOPSLA*, pages 817–832, 2013.

[8] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *POPL*, pages 549–560, 2013.

[9] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *PLDI*, pages 453–462, 2012.

[10] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.

[11] S. Boldo and J.-C. Filliâtre. Formal verification of floating-point programs. In *IEEE ARITH*, pages 187–194, 2007.

[12] R. P. Brent. *Algorithms for Minimization without derivatives*. Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[13] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev. Efficient search for inputs causing high floating-point errors. In *PPOPP*, pages 43–52, 2014.

[14] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[15] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[16] D. Dunbar, C. Cadar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[17] I. A. Espírito-Santo, L. A. Costa, A. M. A. C. Rocha, M. A. K. Azad, and E. M. G. P. Fernandes. On challenging techniques for constrained global optimization. In *Handbook of Optimization*, pages 641–671, 2013.

[18] P. Fitzpatrick. Extending backward error assertions to tolerance of large errors in floating point computations. *IEEE Trans. Computers*, 46(4):505–510, 1997.

[19] Z. Fu. Modularly combining numeric abstract domains with points-to analysis, and a scalable static numeric analyzer for Java. In *VMCAI*, pages 282–301, 2014.

[20] A. Gáti. Miller analyzer for Matlab: A Matlab package for automatic roundoff analysis. *Computing and Informatics*, 31 (4):713–726, 2012.

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

[22] M. Goldstein. Significance arithmetic on a digital computer. *Commun. ACM*, 6(3):111–117, 1963.

[23] E. Goubault. Static analyses of the precision of floating-point operations. In *SAS*, pages 234–259, 2001.

[24] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *SAS*, pages 18–34, 2006.

[25] J. Harrison. Decimal transcendentals via binary. In *IEEE ARITH*, pages 187–194, 2009.

[26] N. J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2nd edition, 2002.

[27] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, March 2012.

[28] D. Jiang and N. F. Stewart. Backward error analysis in computational geometry. In *ICCSA*, pages 50–59, 2006.

[29] T. Kaneko and B. Liu. On local roundoff errors in floating-point arithmetic. *J. ACM*, 20(3):391–398, 1973.

[30] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[31] D. E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. Addison-Wesley Professional, 3 edition, Nov. 1997. ISBN 0201896842.

[32] M. Martel. Semantics-based transformation of arithmetic expressions. In *SAS*, pages 298–314, 2007.

[33] W. Miller and D. L. Spooner. Algorithm 532: Software for roundoff analysis [Z]. *ACM TOMS*, 4(4):388–390, 1978.

[34] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, 2004.

[35] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. 2010.

[36] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.

[37] C. Rubio-González, C. N. 0001, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: tuning assistant for floating-point precision. In *SC*, page 27, 2013.

[38] W. Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, New York, third edition, 1976.

[39] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *PLDI*, pages 53–64, 2014.

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

[41] N. H. Tuan, P. H. Quan, D. D. Trong, and L. M. Triet. On a backward heat problem with time-dependent coefficient: Regularization and error estimates. *Applied Mathematics and Computation*, 219(11):6066–6073, 2013.

[42] D. J. Wales and J. P. K. Doye. Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms. *The Journal of Physical Chemistry A*, 101(28):5111–5116, Mar. 1998.

[43] J. H. Wilkinson. Rounding errors in algebraic processes. In *IFIP Congress*, pages 44–53, 1959.

[44] J. H. Wilkinson. Error analysis of floating-point computation. *Numerische Mathematik*, 2(1):319–340, 1960.

[45] J. H. Wilkinson. Some comments from a numerical analyst. *J. ACM*, 18(2):137–147, 1971.

[46] J. H. Wilkinson. Error analysis revisited. *IMA Bulletin*, 22 (11/12):192–200, 1986.

[47] I. Zelinka, V. Snsel, and A. Abraham. *Handbook of Optimization: From Classical to Modern Approach*. Springer Publishing Company, Incorporated, 2012. ISBN 3642305032, 9783642305030.

[48] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei. A genetic algorithm for detecting significant floating-point inaccuracies. In *The 37th International Conference on Software Engineering*, Firenze, Italy, 2015.

A. Parameters Used in Our Experiments

In this section, we give details on the parameters we have used in our experiments. The objective is to facilitate researchers and developers to reproduce our results. Note, however, that our algorithms are based on a Monte-Carlo process, and it is unlikely to obtain the exact same experimental results as presented in Sect. 4.

Algo. 1 and 2 as implemented in our experiments have several important parameters that are set as follows:

Parameters	Values set in our experiments
ftol	$\Phi_x(0)$*1E-3 where Φ_x is defined in Eq. (16)
xtol	1E-4
cc	0.9
iter_local	100
iter_global	100, 1000 or 10000
n_start	100

Some parameters are more sensitive than others. In particular, to set *ftol* appropriately can be difficult. Recall that, given $f, \hat{f}, x$, the backward error $B(x)$ is defined as the smallest $|\delta|$ so that the formula

$$|f(x+\delta \cdot x) - \hat{f}(x)| \leq ftol \tag{29}$$

holds. If *ftol* is set larger than $|f(x) - \hat{f}(x)|$, formula (29) holds for $\delta = 0$, leading to $B(x) = 0$. If *ftol* is too small, formula (29) is unsatisfiable, and $B(X)$ will be undefined because the search space of the mathematical optimization problem (14) becomes empty in this case. To avoid these issues, in our experiments we set *ftol* strictly smaller than $|f(x) - \hat{f}(x)|$. This way, the case $B(x) = 0$ above can be avoided. Further, to avoid the mentioned case of an undefined $B(x)$, we can set *ftol* to be proportional to $|f(x) - \hat{f}(x)|$ as specified in the table above.