A Comprehensive Study of Real-World Numerical Bug Characteristics

Anthony Di Franco*, Hui Guo*, and Cindy Rubio-González
Department of Computer Science
University of California, Davis, USA
{acdifranco, higuo, crubio}@ucdavis.edu

***Abstract*—Numerical software is used in a wide variety of applications including safety-critical systems, which have stringent correctness requirements, and whose failures have catastrophic consequences that endanger human life. Numerical bugs are known to be particularly difficult to diagnose and fix, largely due to the use of approximate representations of numbers such as floating point. Understanding the characteristics of numerical bugs is the first step to combat them more effectively. In this paper, we present the first comprehensive study of real-world numerical bugs. Specifically, we identify and carefully examine 269 numerical bugs from five widely-used numerical software libraries: NumPy, SciPy, LAPACK, GNU Scientific Library, and Elemental. We propose a categorization of numerical bugs, and discuss their frequency, symptoms and fixes. Our study opens new directions in the areas of program analysis, testing, and automated program repair of numerical software, and provides a collection of real-world numerical bugs.**

I. Introduction

Numerical software provides the foundation for a wide variety of software applications, including safety-critical systems such as control systems for vehicles, medical equipment, and industrial plants. Libraries for machine learning (e.g., TensorFlow, scikit-learn), computer graphics (e.g., OpenGL), computer vision (e.g., OpenCV), and data analysis (e.g., Pandas) rely on numerical libraries such as NumPy [7], SciPy [24], and LAPACK [6] to perform numerical calculations. Domain-specific languages such as GNU Octave for scientific programming and R for statistical computing integrate numerical libraries to provide support for numerical computations.

Numerical software relies heavily on floating point arithmetic, which brings additional challenges in terms of software reliability and performance. Floating point is a widely used representation that approximates real numbers. By nature, floating point introduces imprecision in numerical calculations. Sources of numerical errors include extreme sensitivity to roundoff, floating-point exceptions such as overflow/underflow, and nonreproducibility across machines or even across runs on the same machine. This has led, in part, to numerical bugs that have caused catastrophic failures [2, 4, 8, 41].

Numerical bugs are those related to either the finite approximate representation of real numbers, or to mathematical aspects of the computation. Techniques have been proposed to estimate roundoff error [18, 20, 36], generate inputs that maximize errors [14], or trigger floating-point exceptions [11], and to detect accuracy [9, 12, 45] and instability problems [10, 26]. However, there is a gap of knowledge in understanding the characteristics of numerical bugs in real-world numerical software. What are the most common numerical bugs? How prevalent are these bugs? Are existing tools capable of detecting them? How are such bugs fixed? Are existing program repair tools suitable to fix them? This paper takes a first step towards answering these questions by conducting the first comprehensive study of real-world numerical bug characteristics. The goal of this paper is to study the causes, symptoms, and fixes of numerical bugs in numerical libraries, and provide a *high-level categorization* that can serve as a guide for researchers to develop tools for finding and fixing numerical bugs.

Empirical studies of bug characteristics have been conducted in the past to learn about concurrency bugs [27, 33], performance bugs [23, 40], and error handling bugs [13, 16], among many others. These studies have revealed patterns for both bug detection and bug fixing. To the best of our knowledge, no study thus far has focused on numerical bugs, or has considered inspecting numerical libraries, despite the reputation of being particularly subtle and error prone.

We faced a number of challenges while conducting this study. First, numerical bugs are not as numerous in general-purpose applications, which motivated us to focus our search on numerical libraries, in particular, NumPy, SciPy, LAPACK, the GNU Scientific Library (GSL) [5], and Elemental [37]. Second, examining bugs from these libraries was particularly challenging due to three main reasons: (1) the libraries use distinct version control systems and bug tracking systems (if any), (2) the libraries range from 7 to 25 years old, thus not all bugs are documented in the same manner, (3) the libraries are implemented in several different programming languages (Python, C/C++, and Fortran,) often with several languages involved in the same library. Finally, we found that, in many cases, examining and classifying numerical bugs required significant domain knowledge, and understanding of the code under inspection.

We identify and carefully examine a total of 269 numerical bugs. We propose to categorize numerical bugs into accuracy bugs, special-value bugs, convergence bugs, and correctness bugs. We find that correctness bugs are the most frequent (37%) in our dataset, followed by special-value bugs (28%), convergence bugs (21%), and accuracy bugs (14%). We discuss their characteristics, including common symptoms and fixing

*Both authors contributed equally.

strategies, and present real-world examples. Finally, based on our findings, we discuss new research directions to develop tools to improve the reliability and efficiency of numerical software.

This paper makes the following contributions:

- We identify and examine 269 real-world numerical bugs found across five widely-used numerical libraries: NumPy, SciPy, LAPACK, GNU Scientific Library (GSL), and Elemental (Sections III and IV).
- We present a categorization of numerical bugs, and discuss their symptoms and fixes (Section IV).
- We discuss new directions to test and analyze numerical programs (Section V).

We provide background on floating point in Section II. We discuss related work in Section VI, and conclude in Section VII.

II. Floating Point Preliminaries

Floating point is the most common representation of real numbers. Unfortunately, floating point requires navigating many subtle tradeoffs and inevitably introduces semantics that differ from those of the reals. Indeed, many of the difficulties faced by numerical programmers are inherent in the semantics of floating point. Here we review the IEEE 754 standard [1] for floating point and its semantics.

A **floating-point value** is one of the following:

- A **number** represented as scb^q where s is the sign (± 1), c is the *coefficient* or *significand*, and q is the *exponent*. The *base* or *radix* b is usually 2 (binary), though the standard also describes a base 10 (decimal) representation. The significand is represented in the same base. Binary formats range in size from half precision (16 bits) to octuple precision (256 bits) in power-of-two bit widths. Most computations are carried out in single (32 bit) or double (64 bit) precision in practice.
- The **infinities** $\pm\infty$ representing infinite quantities. The sign of an infinity will correspond to the signs of the operands used to produce it, i.e., $1/0$ and $-1/-0$ yield ∞ while $-1/0$ and $1/-0$ yield $-\infty$.
- The **`NaN` value** representing a result that is not a representable number, such as the result of an attempt to take the square root of a negative number. `NaN`s generally propagate through computations once they occur, and come in *quiet* and *signaling* varieties to implement sensible exception-generating behavior given this propagation.

Because the significand is scaled by a function of the exponent, the absolute precision of representable numbers varies with the exponent over their range. The width of the interval enclosed by a number with its last significand bit set to zero and the same number with its last significand bit set to one is called a *Unit in the Last Place* or ULP.

Overflow, underflow, and subnormal numbers. Results of computations that exceed the greatest representable number in magnitude in the current precision are said to *overflow*, and those that fall between zero and the smallest representable nonzero number are said to *underflow*.

TABLE I
Floating-Point Exception Behavior

Exception	Condition	Default Result
Underflow	Result between smallest normal number and zero	subnormal number
Overflow	Result larger than largest representable number in magnitude	$\pm\infty$
Inexact	Result was not exactly representable	Rounded result
Invalid	Result was indeterminate or not representable as a number	`NaN`
Divide by zero	Finite, nonzero number is divided by zero	$\pm\infty$

A *subnormal number* uses an exponent of zero to indicate that the exponent is the lowest representable and that the significand contains leading zero bits. This convention allows all remaining precision in the significand to be gradually exhausted as representable numbers approach zero.

Roundoff and truncation errors. Converting from a higher to a lower precision requires truncating the representations of the significand and exponent to the lengths allowed by the lower precision, which reduces both the available dynamic range and precision. The loss of precision is called *truncation error* while the loss of dynamic range can result in overflow.

Roundoff error is the consequence of needing to express the result of an operation in terms of representable numbers, which due to varying absolute precision across the representable range and other effects incurs errors even for operations on numbers that are themselves representable. Roundoff error of basic arithmetic operations is specified to be at most one-half of an ULP, though some implementations that favor speed (such as GPUs) violate this requirement.

Floating-point exceptions. Table I shows a summary of the floating-point exceptions. An *underflow exception* occurs when a result is too small to be represented by a normal number, while an *overflow exception* occurs when the result of an operation is too large to be represented. An *inexact exception* occurs when the result of a floating point operation was rounded. An *invalid exception* occurs when an indeterminate form, such as $0/0$ is evaluated. A *divide by zero exception* occurs when division by zero is attempted. The environment may choose to mask exceptions, in which case the appropriate subnormal numbers, infinities or `NaN`s are used to represent the results and no exception is raised.

Catastrophic cancellation. Subtracting two nearly equal numbers means that most of the bits in their representations will cancel, incurring a large risk that the result will denormalize (underflow). The resulting approximation error will then propagate through the rest of the computation.

Two notable techniques intended to address the difficulties in dealing with the many undesired behaviors that can arise in computing with IEEE floating point and to track the error that accumulates in the course of computation are *interval*

arithmetic, standardized as IEEE 1788 in 2015 [22] and *unum arithmetic* [21]. Interval arithmetic represents a compact set of real numbers by its bounds, which allows tracking the set of possible solutions computed by a program, whose size measures the error. Unum arithmetic is related, and uses representations that partition the real line into exactly representable points and open sets between them. In addition to building error tracking into the representations, unum arithmetic improves the semantics of operations involving infinities or `NaN` versus IEEE 754 floating point.

III. Methodology

This section describes the critera for selecting numerical libraries and the bugs to be examined, and the threats to the validity of our study.

A. Selection of Numerical Libraries

We selected five representative numerical libraries for our study: NumPy, SciPy, LAPACK, GSL, and Elemental. These libraries are characterized by their maturity (7 to 25 years old), popularity (used by thousands of projects), and active development. NumPy and SciPy are Python libraries that combine Python numerical code with wrappers for low-level routines written in C and Fortran, targeting general numerical and scientific computing, respectively. LAPACK is a C/Fortran library of low-level numerical routines, and a building block for many higher-level libraries such as NumPy. GSL mainly implements special functions and probability distributions essential to scientific computing.

Finally, we searched for public GitHub C++ repositories using keywords associated with numerical computation. We ranked the results by number of stars (project popularity). The Elemental library was ranked first. Elemental is a library that provides efficient and general linear algebra routines suitable for numerical analysis, scientific computing, and work in theoretical mathematics, via support for features such as arbitrary precision, and large-scale distributed computation.

For each library, Table II shows its language of implementation, the earliest date for which data is available, whether it is hosted in GitHub, bug tracking system used, current size (LOC), number of commits, contributors, and releases.

B. Selection and Characterization of Numerical Bugs

We chose to examine bug reports (as opposed to commits) to have access to more bug information that includes original reports from users and developer discussions. We imposed a few requirements on our bug report selection to include bugs (1) confirmed by developers (e.g., status is *closed*), (2) considered important (e.g., bug fix is available), and (3) more likely to be numerical bugs.

NumPy, SciPy, and Elemental are hosted on GitHub, and use GitHub's integrated issue tracking system. This is often used by developers to track bugs, enhancements, and other tasks. Additionally, it includes GitHub pull requests. Pull requests are used by developers to contribute code (after review and approval) to a repository. LAPACK, by contrast, hosted its own development before version 3.6.1, when it was migrated to GitHub (adopting GitHub's issue tracking system). Thus, there are two sources of bug reports for LAPACK, (1) the netlib page, which lists the bugs filed between LAPACK 3.0 (released in 2000) and LAPACK 3.6.1 (released in 2016); and (2) the GitHub repository's issue tracker since LAPACK 3.6.1. We refer to these as LAPACK1 (before GitHub) and LAPACK2 (after GitHub) through the rest of the paper. Finally, GSL maintains a Savannah bug tracking system.

We found that all libraries, except for LAPACK1 and GSL, make use of labels to classify issues. In our selection, we filtered out any issues with labels related to build issues, documentation issues, and feature requests.

After filtering out open issues, issues without patches, and issues without relevant labels, the number of potentially interesting issues for NumPy and SciPy was still too large for manual inspection. Thus, we applied two additional filters to these projects to further refine our selection. First, we searched issue titles and descriptions for the following keywords: *nan*, *exception*, *overflow*, *underflow*, *infinity*, *infinite*, *precision*, *unstable*, *instability*, *ringing*, *unbounded*, *roundoff*, *truncation*, *rounding*, *diverge*, *cancellation*, *cancel*, *accuracy*, *accurate*. Second, we randomly sampled from the resulting set of issues.

We manually examined the selected bugs by reading their descriptions, comments, and any associated commits or pull requests to verify that they were numerical bugs of interest to our study. Then, we made a second pass through all bugs that passed this final selection criterion, and classified them according to the symptoms they displayed and the strategies used to implement their fixes.

C. Threats to Validity

Internal Validity. Our findings depend entirely on the set of bugs we examine. Thus, we ensured that the relevance of the initial bug selection was as high as possible by applying several filters. First, we made sure the issues and pull requests we selected were real bugs by choosing only accepted/closed issues (except for GSL where, due to limited data, we also examined issues that were open, but manually assured they had been confirmed before including them). Second, when available, we used labels from the project bug tracker as part of the selection criteria to ensure that we examined bugs considered important by project developers. Third, because we are also interested in bug fixing, we selected bugs referenced by other bugs or commits when possible, skewing our selection towards bugs with associated patches. Fourth, if the number of selected issues was too large for manual inspection, we used keywords indicative of numerical problems to select those bugs most likely to be relevant. Finally, each bug was inspected independently by each author of this paper.

External Validity. The results presented in this paper are based on only five numerical libraries, requiring special attention to select a representative sample to make generalization to other bugs in other software justified. To address this challenge, we selected representative, mature, and widely-used numerical libraries. Our selection includes libraries implemented in

TABLE II
NUMERICAL LIBRARY INFORMATION

Library	Language	Start Date	GitHub	Bug Tracker	LOC	# Commits	# Contributors	# Releases
NumPy	Python	Dec 16, 2001	Y	GitHub	15,366	15,731	512	125
SciPy	Python	Jan 28, 2001	Y	GitHub	823,446	17,012	479	91
Elemental	C++	Mar 14, 2010	Y	GitHub	778,156	3,721	23	13
LAPACK1	C/Fortran	May 31, 2000	N	Website	1,613,856	NA	60	17
LAPACK2	C/Fortran	Oct 26, 2008	Y	GitHub	1,776,339	1,249	19	3
GSL	C/C++	Dec 17, 2007	N	Savannah	278,617	5,596	15	21

TABLE III
NUMERICAL LIBRARY ISSUES AND PULL REQUESTS

	Issues					Pull Requests					
Library	All	Closed	w/Patch	Label	Inspected	All	Closed	w/Patch	Label	Inspected	Total Insp
NumPy	5096	3745	1335	682/195	80	3912	3722	3172	431/63	20	100
SciPy	4093	3198	885	561/170	80	3252	3088	2667	153/39	20	100
Elemental	123	85	18	17	85	107	100	85	81	100	185
LAPACK1	148	141	135	-	135	-	-	-	-	-	135
LAPACK2	59	45	-	31	31	81	79	-	59	59	90
GSL	218	116	47	-	218	-	-	-	-	-	218
Total	9737	7330	2420	1291	**629**	7352	6989	5924	724	**199**	**828**

TABLE IV
CATEGORIZATION OF NUMERICAL BUGS

Library	Accuracy	S-Values	Converg.	Correctness	Total
NumPy	5	16	0	3	24
SciPy	8	27	6	30	71
Elemental	0	0	0	9	9
LAPACK	11	11	11	9	42
GSL	14	22	39	48	123
Total	38	76	56	99	**269**

several of the most commonly used programming languages for numerical code and addressing a comprehensive range of subtypes of numerical code at levels of abstraction from low-level arithmetic and linear algebra to higher-level, large-scale scientific computing and number theory.

IV. NUMERICAL BUG STUDY

The main purpose of this numerical bug study is to answer the following research questions:

R1. How frequent are numerical bugs?
R2. How can we group numerical bugs into categories that share common causes and patterns, and how frequently do bugs in each category occur?
R3. What other characteristics of numerical bugs can we identify to inform building tools?

We examined a total of 828 bugs (issues and pull requests) from five numerical libraries (NumPy, SciPy, Elemental, LAPACK, and GSL). Table III lists the total number of issues, and pull requests (PRs) per library. The column "Closed" refers to the number of issues/PRs with status *closed*. The column "w/Patch" indicates the number of closed issues/PRs that include patches, and the column "Label" indicates the number of closed issues/PRs with patches that were selected based on their labels. For SciPy and NumPy we provide two numbers. The second number is the number of labeled issues/PRs in which relevant numerical keywords (see Section III) were found. From these, we randomly selected 80 issues and 20 PRs from each of NumPy and SciPy. We inspected all 218 bug reports for GSL, and all closed 85 issues and 100 PRs for Elemental. In total, we inspected 629 issues and 199 PRs. We use issues/PRs and bugs interchangeably in the rest of this paper.

We found 269 numerical bugs in the 828 bugs inspected, and developed a categorization of numerical bugs that consisted of four groups: *accuracy*, *special value*, *convergence*, and *correctness*. Table IV shows the distribution of each kind of bug across the five libraries studied. Next we describe the characteristics of each bug category.

Finding 1: 32% of the bugs examined are numerical bugs. Based on our observations, we propose four categories for numerical bugs: accuracy, special value, convergence, and correctness.

A. Accuracy Bugs

We classified bugs as *accuracy bugs* when precision loss due to rounding or truncation errors led to an incorrect result. Out of the 269 numerical bugs, we found 38 accuracy bugs across four numerical libraries. These bugs were found mostly in decomposition and eigenvalue computation routines in LAPACK, and in the core components of GSL, Scipy, and NumPy. We found that accuracy bugs often fell into one of three subcategories:

(1) Insufficient Precision Data Type. To determine the precision of a variable one must take into account the range of values to be stored. While using too much precision can cause performance degradation and/or increase in memory usage, using too little precision can cause precision loss that may lead

```
>>> import numpy as n
>>> a = n.ones((1000,1000),dtype=n.float32)*132.00005
>>> a.min()
132.000045776
>>> a.max()
132.000045776
>>> a.mean()
133.96639999999999
```

Fig. 1. NumPy issue #1063: tests revealing accuracy bug due to insufficient precision data type.

```
double complex hyp0f1_cmplx(double v, double complex z):
     ...
+    double complex t1, t2

     # both v and z small: truncate the Taylor series
-    if zabs(z) < 1e-6*(1.0 + fabs(v)):
-        return 1.0 + z/v + z*z/(2.0*v*(v+1.0))
+    if zabs(z) < 1e-6*(1.0 + zabs(v)):
+        t1 = 1.0 + z/v
+        t2 = z*z / (2.0*v*(v+1.0))
+        return t1 + t2
```

Fig. 2. SciPy issue #6368: bug due to an inaccurate arithmetic expression.

to incorrect results. Fig. 1 illustrates an example of an accuracy bug due to insufficient precision data type found in the NumPy library. There, the average of an array is well outside of the range between its minimum and maximum elements. This is due to the accumulation of roundoff errors in the summation of the array elements. The suggested fix proposes to use higher precision for the variable that stores the sum. However, some developers argued that higher precision would be too expensive to use, and the issue was closed without applying the fix. The bug is still present in the current version of the library.

(2) Inaccurate Arithmetic Expressions. Floating-point arithmetic expressions must be carefully crafted to minimize the generation and propagation of roundoff errors. Fig. 2 shows an example where the order in which arithmetic operations are computed leads to precision loss. The bug is fixed by introducing two temporary variables and breaking the original arithmetic expression into three separate expressions to enforce the required order of computation. Fig. 3 shows another example: two expressions that suffer from catastrophic cancellation are replaced with different but equivalent expressions.

(3) Ill-Conditioned Problem. When a large numerical error is found in a result, the issue may be inherent to the problem, as it would appear regardless of the algorithm used. The *condition number* of the problem is the key to make this distinction. A large condition number indicates that the problem itself is sensitive to numerical errors.[1] Fig. 4 shows an example of an ill-conditioned problem. After examining the computation order of an expression as shown in Fig. 2, the developers discussed the problem further and recognized that the problem is ill-conditioned – its condition number is much larger than

[1]The condition number measures how sensitive a result is to small changes in the input. A problem is ill-conditioned when a small change in the input leads to a large change in the output, magnifying the impact of errors.

```
     def _cdf(self, x):
-        return 2 * (1 - _norm_cdf(1 / sqrt(x)))
+        # Equivalent to 2*norm.sf(sqrt(1/x))
+        return special.erfc(sqrt(0.5 / x))

     def _ppf(self, q):
-        val = _norm_ppf(1 - q / 2.0)
+        # Equivalent to 1.0/(norm.isf(q/2)**2)
+        val = -special.ndtri(q/2)
         return 1.0 / (val * val)
```

Fig. 3. SciPy issue #3545: bug due to inaccurate expressions.

```
scipy.special.hyp0f1(v, z)
    Confluent hypergeometric limit function 0F1.

    Parameters:
        v, z : array_like
            Input values.
    Returns:
        hyp0f1 : ndarray
            The confluent hypergeometric limit function.
--------------------------------------------------------
The problem is ill-posed: the condition number is
|dF/dz| * |z|/|F| ~ 1/|v| >> 1 at v=z<<1. Small
relative changes in input values cause large relative
changes in output.
```

Fig. 4. SciPy issue #6368: accuracy bug due to ill-conditioned problem.

1 when v is much smaller than 1. High precision data types would be necessary to achieve correct behavior.

Detection and Fixing Automation. In the bugs we examined, backward error analysis was often used in the detection of accuracy bugs. For example, to test a division on floating-point numbers, $q, r = divmod(a, b)$, the accuracy of the quotient q and remainder r can be determined by comparing $q * b + r$ and a. A similar technique can be used to test decomposition and eigenvalue computation routines. In these cases, the two resulting matrices should be orthogonal in theory. A problem is revealed if their product greatly deviates from the unit matrix. Such techniques could be used for bug detection. Input generation is another important aspect in detecting accuracy bugs, however we did not find any mention of input generation when examining the bug reports. Finally, we found that the first two subcategories suggest strategies for fixing accuracy bugs: switching to higher precision, and transforming arithmetic expressions to improve their precision.

Finding 2: Accuracy bugs are the least frequent numerical bugs in our dataset (14%). Techniques such as backward error analysis could be used to automatically detect these bugs. Furthermore, we identify two common strategies to fix accuracy bugs: using higher precision, and reordering arithmetic expressions. Input generation to detect accuracy bugs remains a challenge.

B. Bugs Related to Special Values

In this paper, we refer to signed zero, subnormal numbers, infinities, and `NaN`s (Not a Number) as *special values*. Unlike normalized floating-point numbers, special values are given a special encoding in the floating-point representation system. Moreover, arithmetic operations on special values follow irregular rules with subtle implications. We denote a bug as

```
>>> import numpy as np
>>> np.max(np.array([-1, np.nan, -2]))
-2
```

Fig. 5. NumPy issue #1511: test revealing a NaN bug in function max.

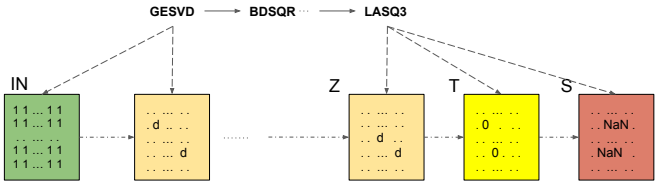

Fig. 6. LAPACK bug #97: an example of a special-value bug. The initial matrix IN has all elements set to 1, the singular value decomposition routine of LAPACK, GESVD eventually produces a matrix that contains NaNs.

related to special values when its root cause is related to computation involving special values. For example, the program fails with NaN inputs, or it allows precision loss to proceed beyond the appearance of subnormal numbers and producing NaNs later in the computation. We found that 76 out of 269 numerical bugs are related to special values. These bugs are distributed widely in different library components, and can cause an algorithm to fail in various ways: producing NaN values, producing incorrect results, causing infinite loops, etc. Next, we describe two kinds of bugs in this category.

(1) Missing NaN Checks. A NaN checking procedure checks the floating-point inputs explicitly for NaN values. Depending on the program's policy to deal with NaNs, inputs that contain NaNs are either rejected, ignored, or masked. Some disagreement is observed among library developers with respect to whether NaN checks should be enabled or not for certain inputs. While enabling NaN checks makes algorithms more robust, these also add runtime overhead. As a result, NaN checks are often added until a function is reported to be buggy on NaN inputs. Fig. 5 shows a test that reveals such a bug in the max function from NumPy. The function returns an incorrect result when the input array contains a NaN value. In the example, the function returns -2 as the maximum value in the array $[-1, np.nan, -2]$. This is due to the fact that any comparison against a NaN evaluates to false. In this case, max checks whether -1 is larger than NaN, which evaluates to false, thus NaN is larger. Then, it checks whether NaN is larger than -2, which also fails, and -2 is returned. To fix this bug, a NaN check is added to check the array for NaNs. Furthermore, the function is modified to immediately return a NaN if the input array contains a NaN value.

(2) Overflow/Underflow. Another important type of special-value bug is overflow/underflow, which can lead to NaNs. These bugs are caused by a variety of problems, and often require domain specific knowledge for their detection. Fig. 6 describes an underflow bug in the LAPACK library that results in NaNs. A user reported NaN values in the result when calling routine GESVD to compute the singular value decomposition of a matrix with all elements set to 1. The input matrix leads to denormalized numbers when bidiagonalized in GESVD. This matrix is eventually passed to routine LASQ3. In LASQ3, T is computed by taking half of $(Z(nn-7)-Z(nn-3))+Z(nn-5)$, where nn is a local scalar, underflowing and becoming 0. NaN values are then produced in S when dividing T. A fix for this bug is proposed by strengthening the conditions for further computation on T, i.e., avoiding the computation of S when T underflows. This fix required domain specific knowledge.

Detection and Fixing Automation. NaNs are designed to denote the results of "invalid" floating-point operations, e.g., $0/0$, without interrupting execution. Furthermore, NaNs are often used to represent missing data, which can lead to ambiguity. Both uses cause errors due to the idiosyncratic behaviors of computations on NaNs and other special values, and their non-local propagation. An approach to detect special-value bugs is to feed NaN inputs to functions to observe where and how they subsequently fail. This could be used to reveal problems with NaN checks or their absence. On the other hand, generating inputs to reveal underflow/overflow errors is, however, difficult to automate, as it requires domain-specific knowledge. Finally, the main challenge in automating bug fixing is to determine when to check for special values, and what to do in their presence.

> **Finding 3**: We find that 28% of the numerical bugs are special-value bugs. These bugs mainly involve NaN checks and overflow/underflow conditions. An initial approach to detect missing NaN checks is to use NaN inputs. Generating inputs to expose overflow/underflow remains a challenge.

C. Convergence Bugs

We define *convergence bugs* as bugs where an iterative or series approximation diverges or converges too slowly due to numerical problems such as magnified roundoff or truncation errors. This is related to numerical instability, i.e., the output is sensitive to small variations in inputs or intermediate quantities. Consequently, small errors introduced by floating-point approximations drive the output far from the correct value. This can be complicated by interactions with instability around conditional tests, where vanishingly small differences in a quantity involved in a test can cause different control paths to be followed. An example is the termination of a loop that intends to continue an iterative approximation until an error estimate goes below a given error tolerance.

We find that 56 out of 269 numerical bugs correspond to convergence bugs; 39 were found in GSL, 11 in LAPACK, and 6 in SciPy. We did not find any convergence bugs in the sampled issues from NumPy or Elemental. This was unremarkable in the case of NumPy because NumPy focuses mainly on low-level linear algebra data structures, and operations on these are delegated to wrapped algorithms from other packages. Elemental includes support for arbitrary-precision arithmetic. Arbitrary-precision arithmetic avoids convergence issues by scaling precision according to the demands at each point in a computation.

Convergence bugs fall into one of the following scenarios:

- The approximation algorithm uses a problematic approximation formula that yields a wrong result.
- Numerical issues in an iterative approximation cause an infinite loop that leads to a program crash.
- Special values such as `NaN`s are involved in divergent behavior. We classify this as a convergence problem yielding a wrong result rather than a special-value issue because it is more specific to do so.

An example of a convergence bug is illustrated in Fig. 7. A user noticed an error larger in magnitude than the result itself in the output of the hypergeometric special function in the regime of large negative a, positive b, and large positive x parameters. Another user noted that the Kummer transformation used for $x < 100$ returned correct results for larger x as well, and proposed a fix to use it by default in this regime for all values of x, which was accepted.

In general, the symptom revealing a convergence bug was usually the output of wrong results. Occasionally, failure to converge would result in a non-terminating loop, hanging the program, which we classified as a crash.

Detection and Fixing Automation. Because failures to converge are due to issues in the quality of a numerical approximation, these require domain-specific knowledge to address, and fixes usually involve using a better approximation technique. In principle, running programs with arbitrary-precision arithmetic could help to automate finding such issues. Since these bugs are among the most common in scientific computing packages that implement special functions, and probability distributions, such a tool could be highly valuable. Unum arithmetic could be especially interesting in this context because it combines aspects of interval-based error tracking with adaptive precision.

Finding 4: We found that 21% of the numerical bugs are convergence bugs. Diagnosing convergence bugs requires domain-specific knowledge. Adaptive use of interval and arbitrary-precision should be investigated further. In this context, Unum arithmetic is especially interesting since it combines a form of interval-based error tracking with adaptive precision.

D. Correctness Bugs

A *correctness bug* is caused by any error in the implementation of an algorithm that have to do primarily with its mathematical or algorithmic structure. Correctness bugs are the most type of numerical bugt; 99 out of 269 numerical bugs are correctness bugs. Correctness bugs have the greatest variety of all types, and ranged from typographical errors in transcribing a formula from a reference, to using an approximation formula for a function outside the function's domain of definition, to errors due to compiler optimizations violating the assumed semantics of mathematical operations.

To differentiate algorithmic correctness bugs that are specific to numerical code from correctness bugs in general-purpose code, we considered only bugs dealing with mathematical aspects of the computation, or at least with the use of data types

```
int gsl_sf_hyperg_1F1_e(const double a, const double b,
                        const double x,
      gsl_sf_result * result) {
    ...
-   else if(a < 0.0 && fabs(x) < 100.0) {
+   else if(a < 0.0 && fabs(x) < 2*GSL_LOG_DBL_MAX) {
      /* Use Kummer to reduce it */
      /* to the generic positive case... */
      gsl_sf_result Kummer_1F1;
      int stat_K = hyperg_1F1_ab_pos(b-a, b, -x,
                    &Kummer_1F1);
      int stat_e = gsl_sf_exp_mult_err_e(x,
                    GSL_DBL_EPSILON * fabs(x),
                    Kummer_1F1.val,
                    Kummer_1F1.err,
                    result);
      return GSL_ERROR_SELECT_2(stat_e, stat_K);
    }
    ...
}
```

Fig. 7. GSL issue #28718: an example of a convergence bug.

or structures specific to numerical computation. For example, an error in a branch condition that resulted in the use of the wrong approximation formula for a special function at the given point in its domain would be classified as a numerical correctness bug, while an error in a branch condition that resulted in the premature return of an uninitialized value would not be counted as a numerical correctness bug. For another example, a concurrency bug arising in distributing computation over a matrix that considers the structural features of that matrix would be considered a numerical correctness bug, while a concurrency bug in general would not be.

Correctness bugs were most prevalent in SciPy, and GSL. We observe that SciPy and GSL both focus on implementing special functions and probability distributions used for scientific computing, and the numerical correctness bugs we found in them occur overwhelmingly in these features.

We classified correctness bugs into the following subcategories:

- Errors in the initialization or updates in iterative solvers that resulted in problems other than a failure to converge to the correct result.
- Errors in conditional tests that resulted in the wrong function approximation being used for the given point in the domain.
- Errors involving vector and matrix data data types, such as element type errors and incorrect array dimension.
- Violations of assumed numeric semantics due to compiler optimizations.

Fig. 8 shows an example of a numerical correctness bug related to error initialization found in SciPy. The bug is found in code wrapping a call to a Fortran routine for Sequential Least-Squares Programming, where there is an erroneous attempt to use dummy values of $\pm 1e12$ for constraint bounds passed to the solver. This resulted in scaling problems in the solver due to the large range of the constraints and also precluded finding solutions outside those bounds even when they existed in the given problem. The underlying Fortran routine in fact supported using `NaN` in the place of bounds for unbounded

```
    def _minimize_slsqp(...):
        ...
-       # filter -inf, inf and NaN values
+       # Mark infinite bounds with nans;
+       # the Fortran code understands this
        infbnd = ~isfinite(bnds)
-       xl[infbnd[:, 0]] = -1.0E12
-       xu[infbnd[:, 1]] = 1.0E12
+       xl[infbnd[:, 0]] = np.nan
+       xu[infbnd[:, 1]] = np.nan
```

Fig. 8. SciPy pull request #6024: an example of a correctness bug.

problems, and once that fix was made (and an array bounds error introduced by the initial fix was then addressed) the routine worked as expected.

Detection and Fixing Automation. Correctness bugs were most often discovered through the symptom of an incorrect result being returned. The majority of these were discovered by users noticing incorrect results in the course of their work and comparing the library results to other implementations of the same feature, usually in less efficient but more rigorous packages that include such features as arbitrary-precision arithmetic, for example PARI/GP[3]. This suggests that differential testing against such a package could be an effective bug-finding approach. A minority were discovered through test cases failing. These were often platform or build specific, and were the fault of the compiler violating semantic assumptions implicit in the code under certain conditions. This suggests the approach of using a database of platform, build, and environment dependent behavior to scan code for instances of patterns that differ in their behavior across targets.

In general, correctness bugs required the most domain-specific knowledge of all types to fix, given that they were most related to the mathematical structure of the problem at hand and the idiosyncrasies of the language and environment.

Finding 5: Correctness bugs are the most common numerical bugs, with 37% of numerical bugs falling into this category. Correctness bugs are extremely challenging to detect. Differential testing may be a good method to detect these bugs.

E. General Bug Symptoms and Opportunities for Automation

We identified three common symptoms of numerical bugs: *wrong results*, *crashes*, and *bad performance*. Table V shows their distribution across libraries. We found that the majority of numerical bugs (77%) are revealed by producing wrong results. This is followed by crashing the program (13%), and causing bad performance (10%). For all libraries except Elemental, producing wrong results was the most common symptom of a numerical bug. The second most common symptom was program crashes for NumPy and SciPy, and bad performance for Elemental[2], LAPACK, and GSL.

We also classified bugs according to whether detection or fixing could be automated. Table VI shows our classification, and the distribution across libraries. "Pattern" describes bugs that could be detected or fixed by applying a simple pattern, "Analysis" refers to bugs that could be potentially detected or fixed automatically after non-trivial analysis of the code, and "Domain Specific" are bugs that require domain specific knowledge to be detected or fixed. We found that very few bugs (4%) can be detected using simple patterns (e.g., matching small AST fragments to repair problematic order of operations). However, we found that a considerable number of bugs (29%) might be detected automatically with program analysis techniques (e.g., dataflow analysis to aid in estimating output error from intermediate roundoff errors). In the case of automated bug fixing, fewer bugs could be fixed by applying patterns (2%), or analyzing code (17%); a large majority of fixes were domain specific. Note that we considered only 183 bug fixes in our study.

[2]In the case of Elemental, this difference may be due in part to its use of arbitrary precision arithmetic, which tends to convert precision loss problems into performance problems by dynamically extending the representation length on demand.

TABLE V
BUG SYMPTOMS

Library	Wrong Results	Crashes	Performance	Total
NumPy	19	4	1	24
SciPy	51	19	1	71
Elemental	0	4	5	9
LAPACK	30	2	10	42
GSL	106	7	10	123
Total	206	36	27	269

Finding 6: The most common symptom for numerical bugs is wrong results, which is observed in 77% of the bugs. This is followed by crashes with 13%, and bad performance with 10%. About 80% of the numerical bugs required domain specific knowledge to be fixed.

V. LESSONS LEARNED

(1) Frequency of Numerical Bugs. Even in the projects we examined, which focus exclusively on numerical code, 32% of the reported issues were numerical bugs. We speculate that this may be due to the relative maturity of these projects, and that the essential functionality is already in place and thoroughly debugged. Because of the widespread use and maturity of these libraries, and their tendency to be implemented by highly-skilled experts, we would also expect that the bugs most relevant to this study were discovered and fixed early in the project lifecycle, if they appeared at all, thus we may need to look for historical bug data that far predates the migration of these projects to GitHub, if they appear on GitHub at all. Finally, numerical bugs can also be present in clients of numerical libraries, and other applications (e.g., [28]). It would be interesting to investigate whether our findings apply to code other than numerical libraries.

(2) Current Tool Usage. We did not find any indication during our manual inspection that bug detection tools were used by developers or users. In most cases, users seemed to find inputs that exposed unexpected behavior almost by chance. There was also no indication that developers of these libraries

TABLE VI
OPPORTUNITIES FOR AUTOMATION

	Bug Detection				Bug Fixing			
Library	All	Pattern	Analysis	Domain Specific	All	Pattern	Analysis	Domain Specific
NumPy	24	3	21	0	20	1	9	10
SciPy	71	1	9	61	68	1	2	65
Elemental	9	0	4	5	9	0	3	6
LAPACK	42	1	14	27	42	0	13	29
GSL	123	7	29	87	44	2	5	37
Total	269	12	77	180	183	4	32	147

had used any existing tools to test the libraries. As future work, it would be interesting to find whether specific existing tools (e.g., tools that find divergences [15]) are able to find the bugs we identified in this study.

(3) Usage of Patterns. Although we identified a small number of bugs that could be detected or fixed by applying simple patterns such as expression rewriting, we believe that there is still promise on applying such patterns to other code bases, in particular, to the numerous clients of these libraries. Also, we could apply those patterns to detect unknown bugs in the libraries considered in this study. Tools such as Herbie [36] and Precimonious [38] could be applicable in these scenarios.

(4) Testing and Input Generation. In our manual inspection, we found a few users who determined that a library was producing an incorrect result because they had performed the same numerical operation in a more rigorous but less efficient library, and found a mismatch in the results. This suggests that differential testing of numerical libraries could be particularly beneficial for numerical software for which oracles are often difficult to obtain. This would be particularly useful to detect many of the bugs that we classified as requiring domain-specific knowledge (the majority in this paper). Even in the absence of a known-good implementation, differential testing of numerical libraries would help to detect many of the most difficult bugs by simply finding mismatching results across different numerical libraries, taking advantage of the tendency of different implementations to fail in different ways in challenging tasks. We also found that techniques for input generation (e.g., [14, 45]) are in great need to uncover many of the problems described throughout this paper.

(5) Program Analysis. Program analysis tools could be developed to automatically detect a good number of numerical bugs, including special-value bugs, accuracy bugs, and convergence bugs. In particular, current challenges in the estimation of roundoff errors (e.g., treatment of loops in [17, 42]) present interesting problems that will have a great impact on being able to find a variety of numerical bugs. In the case of dynamic tools, the main challenge will be to produce results that can be generalized to many program runs while imposing low runtime overhead. A delicate balance between static and dynamic approaches is needed.

(6) Automated Bug Fixing. As with other kinds of bugs, automated bug fixing of numerical bugs seems to be the most challenging. The first step would be to apply the state-of-the-art (e.g., [31, 32, 44]) in automated bug repair to see if any of the bugs we identified could be fixed. For certain narrowly-defined cases, simple pattern matching may provide some low-hanging fruit in the meantime.

VI. RELATED WORK

Empirial Studies. Previous empirical studies have examined a variety of software systems, and characterized their defects and associated information on various dimensions [19, 23, 33, 34, 43]. We summarize some of the most related studies that have inspired our work. A study of performance bugs [23] examined 109 performance bugs from Apache, Chromium, GCC, Mozilla, and MySQL, classifying them according to their root cause, how the bug was introduced, how it was exposed, how it was fixed, and its location in the code. A more recent study [40] examined 98 fixed performance bugs in 16 popular client-side and server-side JavaScript projects, finding eight common root causes. In our study, we found that about 10% of numerical bugs have performance-related symptoms. To the best of our knowledge, previous studies on performance bugs did not find instances of numerical bugs causing performance problems.

Similar studies have focused on other aspects of software. For example, Dietz et al. [19] investigate integer overflow bugs in C/C++ programs by performing dynamic checking on the SPEC CINT 2000 benchmarks and also identifying undefined integer overflows in widely used open source software. Our study instead investigates floating-point numerical bugs, and in addition to overflow (and underflow) develops a comprehensive categorization of bugs empirically. In recent studies (e.g., [29, 30, 35]), the authors have developed tools to address specific bug types using pattern-matching based detection and correction techniques informed by empirical analyses, which in part motivated our empirical study.

Tools for Numerical Code. A variety of tools have been developed to improve the reliability and performance of numerical applications. For example, techniques have been devised to estimate roundoff errors. FPTaylor [42] focuses on providing a tight overapproximate estimate on roundoff errors by rigorously handling the transcendental functions using Symbolic Taylor Expansions. Darulova and Kuncak [17] present a programming model that provides real data type for programmers and guarantees a sound compilation to finite-precision implementation that meets the desired precision.

At the same time, a series of dynamic tools have been developed to test and optimize numerical code. For example, techniques [14, 45] have been proposed to generate inputs that lead to large roundoff errors that can be used to test numerical code. Tools for precision tuning [25, 38, 39] dynamically search over the types of variables or instructions to find a mixed precision configuration that (1) produces an accurate result with respect to an error threshold, and (2) improves performance. Herbie [36] rewrites floating-point arithmetic expressions with mathematically equivalent alternatives in order to improve the accuracy of the computations.

Though promising, the above-mentioned tools, as many others, remain narrow in scope and have not been adopted by real-world applications. We hope that our study on real-world numerical bugs can provide context to improve existing tools and to inform the design of future tools so that they engage better with the full depth and breadth of numerical programming problems in the real world.

VII. Conclusions

This paper presented the first comprehensive study of real-world numerical bug characteristics. We examined 828 issues and pull requests from a diverse set of numerical libraries: NumPy, SciPy, Elemental, LAPACK, and GSL. From these, we identified and carefully examined 269 numerical bugs. We found that numerical bugs can be largely categorized into four groups: accuracy bugs, special-value bugs, convergence bugs, and correctness bugs. Correctness bugs, with 37%, comprise the single most common category, and are also the ones that require the most expertise. Correctness bugs are followed by special-value bugs, convergence bugs, and accuracy bugs with 28%, 21%, and 14%, respectively.

We discussed the characteristics of numerical bugs. We found that the most common symptom for numerical bugs are wrong results, followed by crashes and bad performance. We discussed the opportunities to automate detection and fixing of numerical bugs. While merely using patterns applies to few bugs, there is still promise in these being applicable to a large number of libraries and library clients. We found that many bugs could potentially be detected using program analysis, and that there may be opportunities in the area of automated bug fixing as well. As for existing tools, we did not find evidence that suggests the current use of testing/analysis tools by library developers or users. Thus, more work is also required to facilitate the adoption of numerical tools in the real world.

VIII. Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was supported in part by NSF grant CCF-1464439, and a Hellman Fellowship.

IX. References

[1] *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, New York, 1985. Note: Standard 754–1985.

[2] Toyota: Software to blame for Prius brake problems. http://www.cnn.com/2010/WORLD/asiapcf/02/04/japan.prius.complaints/index.html. Accessed: 2017-01-15.

[3] *PARI/GP, version* `2.9.0`. Bordeaux, 2017. URL http://pari.math.u-bordeaux.fr/.

[4] The Explosion of the Ariane 5. https://www.ima.umn.edu/~arnold/disasters/ariane.html. Accessed: 2017-01-15.

[5] GSL - GNU Scientific Library. https://www.gnu.org/software/gsl/. Accessed: 2017-05-01.

[6] LAPACK – Linear Algebra PACKage. http://www.netlib.org/lapack/. Accessed: 2017-05-01.

[7] NumPy: Base N-Dimensional Array Package. http://www.numpy.org/. Accessed: 2017-05-01.

[8] *The Wall Street Journal*, page 37, November 1983.

[9] D. An, R. Blue, M. Lam, S. Piper, and G. Stoker. Fpinst: Floating point error analysis using dyninst, 2008. URL http://www.freearrow.com/downloads/files/fpinst.pdf.

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

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

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

[13] N. Cacho, E. A. Barbosa, J. Araujo, F. Pranto, A. F. Garcia, T. César, E. Soares, A. Cassio, T. Filipe, and I. García. How does exception handling behavior evolve? an exploratory study in java and c# applications. In *ICSM*, pages 31–40, 2014.

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

[15] W. Chiang, G. Gopalakrishnan, and Z. Rakamaric. Practical floating-point divergence detection. In *LCPC*, pages 271–286, 2015.

[16] R. Coelho, L. Almeida, G. Gousios, and A. van Deursen. Unveiling exception handling bug hazards in android based on github and google code issues. In *MSR*, pages 134–145, 2015.

[17] E. Darulova and V. Kuncak. Sound compilation of reals. In *POPL*, pages 235–248, 2014.

[18] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *FMICS*, pages 53–69, 2009.

[19] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in c/c++. In *ICSE*, pages 760–770, 2012.

[20] Z. Fu, Z. Bai, and Z. Su. Automated backward error analysis for numerical code. In *OOPSLA*, pages

639–654, 2015.

[21] J. L. Gustafson. Beyond floating point: Next-generation computer arithmetic, 2017. URL http://web.stanford.edu/class/ee380/Abstracts/170201-slides.pdf.

[22] IEEE. *1788-2015 — IEEE Standard for Interval Arithmetic*. June 2015. doi: https://doi.org/10.1109/IEEESTD.2015.7140721. Approved 11 June 2015 by IEEE-SA Standards Board.

[23] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88, 2012.

[24] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL http://www.scipy.org/. [Online; accessed 2017-05-01].

[25] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *ICS*, pages 369–378, 2013.

[26] W. Lee, T. Bao, Y. Zheng, X. Zhang, K. Vora, and R. Gupta. RAIVE: runtime assessment of floating-point instability by vectorization. In *OOPSLA*, pages 623–638, 2015.

[27] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ASPLOS*, pages 517–530, 2016.

[28] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zähl, and K. Wehrle. Floating-point symbolic execution: A case study in N-version programming. In *ASE*, 2017.

[29] Y. Lin and D. Dig. Check-then-act misuse of java concurrent collections. In *ICST*, pages 164–173, 2013.

[30] Y. Lin, S. Okur, and D. Dig. Study and refactoring of android asynchronous programming (t). In *ASE*, pages 224–235, 2015.

[31] F. Long and M. Rinard. Staged program repair with condition synthesis. In *FSE*, pages 166–178, 2015.

[32] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *POPL*, pages 298–312, 2016.

[33] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, 2008.

[34] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side javascript bugs. In *ESEM*, pages 55–64, 2013.

[35] S. Okur, D. L. Hartveld, D. Dig, and A. v. Deursen. A study and toolkit for asynchronous programming in c#. In *ICSE*, pages 1117–1127, 2014.

[36] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI*, pages 1–11, 2015.

[37] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw.*, pages 13:1–13:24.

[38] C. Rubio-González, C. Nguyen, 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*, pages 27:1–27:12, 2013.

[39] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough. Floating-point precision tuning using blame analysis. In *ICSE*, pages 1074–1085, 2016.

[40] M. Selakovic and M. Pradel. Performance issues and optimizations in javascript: an empirical study. In *ICSE*, pages 61–72, 2016.

[41] R. Skeel. Roundoff error and the patriot missile. *SIAM News*, 25(4):11, 1992.

[42] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM*, pages 532–550, 2015.

[43] A. Vahabzadeh, A. M. Fard, and A. Mesbah. An empirical study of bugs in test code. In *ICSME*, pages 101–110, 2015.

[44] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.

[45] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei. A genetic algorithm for detecting significant floating-point inaccuracies. In *ICSE*, pages 529–539, 2015.