Discovering Discrepancies in Numerical Libraries

Jackson Vanover
University of California, Davis
United States of America
jdvanover@ucdavis.edu

Xuan Deng
University of California, Davis
United States of America
xudeng@ucdavis.edu

Cindy Rubio-González
University of California, Davis
United States of America
crubio@ucdavis.edu

ABSTRACT

Numerical libraries constitute the building blocks for software applications that perform numerical calculations. Thus, it is paramount that such libraries provide accurate and consistent results. To that end, this paper addresses the problem of finding discrepancies between synonymous functions in different numerical libraries as a means of identifying incorrect behavior. Our approach automatically finds such synonymous functions, synthesizes testing drivers, and executes differential tests to discover meaningful discrepancies across numerical libraries. We implement our approach in a tool named FPDiff, and provide an evaluation on four popular numerical libraries: GNU Scientific Library (`GSL`), `SciPy`, `mpmath`, and `jmat`. FPDiff finds a total of 126 equivalence classes with a 95.8% precision and 79.0% recall, and discovers 655 instances in which an input produces a set of disagreeing outputs between function synonyms, 150 of which we found to represent 125 unique bugs. We have reported all bugs to library maintainers; so far, 30 bugs have been fixed, 9 have been found to be previously known, and 25 more have been acknowledged by developers.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Software reliability**; **Empirical software validation**.

KEYWORDS

software testing, numerical libraries, floating point, differential testing

ACM Reference Format:
Jackson Vanover, Xuan Deng, and Cindy Rubio-González. 2020. Discovering Discrepancies in Numerical Libraries. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3395363.3397380

1 INTRODUCTION

Science and industry place their faith in numerical software to give accurate and consistent results. Numerical libraries make up the building blocks for such software, offering collections of *discrete numerical algorithms* that implement *continuous analytical mathematical functions*. In an effort to establish a trusted foundation from which to build powerful and useful tools, developers of numerical libraries aim to offer a certain level of correctness and robustness in their algorithms. Specifically, a discrete numerical algorithm should not *diverge* from the continuous analytical function it implements for its given domain.

Extensive testing is necessary for any software that aims to be correct and robust; in all application domains, software testing is often complicated by a deficit of reliable test oracles and immense domains of possible inputs. Testing of numerical software in particular presents additional difficulties: there is a lack of standards for dealing with inevitable numerical errors, and the IEEE 754 Standard [1] for floating-point representations of real numbers inherently introduces imprecision. As a result, bugs are commonplace even in mature and widely-used pieces of numerical software [19], thus motivating the exploration of analysis and testing techniques.

With regard to the missing-oracle problem in the context of floating-point functions, a common technique involves taking the original function, F, allocating more bits per numerical variable to yield a "higher-precision" function F^*, and treating the output of F^* as the ground truth [9, 15, 16, 20, 42, 48, 49]. Such works rely on the assumption that F^* will have fewer *points of divergence* from the continuous analytical function, call it F_{exact}, and can thus be trusted as an approximation of F_{exact}. However, work by Wang et al. [44] regarding *precision-specific operations* illustrates the conditions under which this assumption fails, resulting in F^* potentially diverging even further from F_{exact}. Additionally, the execution of F^* can become prohibitively expensive: Benz et al. [9] report an increase in the time-to-complete that ranges from a factor of 167× to 1016× slowdown. Even in a situation in which F^* satisfies the above assumption, any points of divergence inherent to the algorithm itself will be present in both F and F^*. Comparing against the same discrete numerical algorithm, albeit at a higher precision, inherently limits the scope of discoverable bugs.

Consequently, this paper proposes a framework for automated and systematic differential testing as an orthogonal approach to complement the existing state-of-the-art. Our approach does not require a ground truth, and therefore, has a chance to uncover a larger variety of bugs. Rather than attempting to compare some numerical function F_1 to F_{exact} or some approximation thereof, our approach compares n numerical functions, $F_1, ..., F_n$, that all implement F_{exact}. We refer to such functions as *function synonyms*. Points of divergence found among these function synonyms and the discrepancies they cause are then reported.

While differential testing is a common technique that has been applied to many domains (e.g., [13, 14, 18, 31, 34, 40, 46]) such testing has not been systematically applied in the context of numerical software. In order to effectively implement differential testing in this domain and report meaningful points of divergence, we have identified three key challenges that must be addressed:

Challenge 1: Automatically finding function synonyms. Determining which functions implement the same continuous analytical mathematical function is non-trivial. Documentation can be unreliable and function names can also be misleading, e.g., the `sinc` function from `mpmath` is *not* a function synonym for the `sinc` functions from either `GSL` or `SciPy`, though `mpmath`'s `rf` is in fact a function synonym for `GSL`'s `poch`. Static approaches to discovering such synonyms in other domains [11, 17, 35, 36] suffer from a lack of parallel corpora or are based on calling context which, for numerical functions, can be widely-varied or even non-existent if being used for one-off calculations. This domain therefore calls for a dynamic approach.

Challenge 2: Extracting signatures and synthesizing drivers. Both finding function synonyms and testing them requires synthesizing drivers to evaluate such functions. Specifically, function signatures must be automatically extracted and each function must be wrapped in an executable driver. While the code for libraries written in statically-typed languages can be easily parsed to gather the information required for synthesizing such a driver, libraries written in dynamically-typed languages present unique difficulties in determining the types of function arguments. Again, documentation can be lacking and non-uniform; an alternative approach is required.

Challenge 3: Identifying meaningful numerical discrepancies. Because of the inherent imprecision of floating-point representations of real numbers, it is often the case that evaluating a pair of function synonyms over the same input will result in two outputs that, while not exactly equivalent, are not indicative of any buggy behavior. How do we know whether or not to call two outputs "equivalent"? Furthermore, the lack of overarching standards with respect to error-handling means that different libraries can produce different outputs when encountering the same erroneous computation. Such discrepancies often stem from deliberate developer choices and are not bugs. To better understand the reported points of divergence, a methodology to identify meaningful discrepancies that represent reportable bugs must be developed.

In this paper, we describe and implement FPDiff, a tool for finding discrepancies between numerical libraries that addresses the above challenges. Given the source code of two or more libraries, FPDiff starts by automatically extracting function signatures (Section 3.1.1) that fit our testing criteria (Section 2.3). To infer parameter types for functions from dynamically-typed libraries, FPDiff leverages developer-written tests. Specifically, FPDiff examines the Abstract Syntax Tree (AST) of test programs to infer argument types for each function call. These extracted function signatures are then used to automatically synthesize drivers to execute the functions (Section 3.1.2). When only *partial* type information is available, the space of possible parameter types is explored by generating multiple drivers, each with a unique configuration of data types.

FPDiff then performs a classification step using the automatically-created drivers to evaluate each function over a set of *elementary inputs* that are designed to result in well-defined behavior (Section 3.2). Functions whose outputs consistently fall within the same $\mathcal{E}$-neighborhood are determined to be function synonyms and placed in the same equivalence class.

This is followed by differential testing (Section 3.3) in which FPDiff executes the drivers of function synonyms in each equivalence class on a diverse set of *adversarial inputs* designed to trigger divergence. Our trials include inputs that are manually crafted by developers, those that are found via binary guided random testing over a subset of the domain to find inputs that maximize inaccuracies in the output, and a set of inputs containing special values that are defined in the IEEE 754 Standard. The outputs across libraries are then compared to identify the points of divergence between function synonyms.

Finally, FPDiff analyzes and reports numerical discrepancies (Section 3.3.2). Each of the resulting discrepancies is placed into one of six categories that are defined in Table 1. We developed this categorization based on our observations of the discrepancies generated by FPDiff, their characteristics, and their likelihood of representing buggy behavior. Finally, a reduction of the set of discrepancies is automatically performed in order to facilitate manual inspection for bugs.

We perform an experimental evaluation over four numerical libraries: The GNU Scientific Library (`GSL`) [24] written in C, the JavaScript library `jmat` [43], and the Python libraries `mpmath` [26] and `SciPy` [27]. All of these libraries were chosen because they are open source, have overlapping functionality, and are widely used in practice. FPDiff finds 655 unique discrepancies between these libraries and, of the reduced set of 327 discrepancies, we found 150 that represented 125 unique bugs. We have reported all bugs to library maintainers; so far, 30 bugs have been fixed, 9 have been found to be previously known, and 25 more have been acknowledged by developers.

This paper makes the following contributions:

- We describe and implement FPDiff, a tool for finding discrepancies between numerical libraries (Section 3).
- We propose a categorization of numerical discrepancies found across numerical libraries (Section 3.3.2).
- We present an evaluation on four popular numerical libraries: `GSL`, `mpmath`, `SciPy`, and `jmat` that finds a total of 655 unique discrepancies, 150 of which represent 125 bugs (Section 4).

The rest of this paper is organized as follows: Section 2 presents a motivating example, preliminary definitions, and describes the scope of our approach. Section 3 details the technical approach to each of the components that make up FPDiff and Section 4 presents an experimental evaluation for each. We discuss related work in Section 5, and conclude in Section 6.

2 PRELIMINARIES

2.1 Illustrative Example

Reasoning about what is the ground truth when it comes to non-trivial floating-point computations is difficult. Short of deriving an analytical solution ourselves or consulting tables of analytically-derived results, true oracles are scarce. Checking against another function that claims to perform the same calculation is a possible strategy to investigate a suspicious result; such output comparisons illustrating unexpected or incorrect behavior are commonly found in bug reports related to numerical functions.

As a motivating example, consider *Tricomi's confluent hypergeometric function*, which provides a solution to Kummer's differential equation and has wide practical applications from pricing stock options [10] to calculating electron velocity in high-frequency gas discharge [32]. Using version 1.3.1 of the numerical library SciPy to compute the value of this function for the input values 0.0001, 1.0, and 0.0001, we see the following:

```
>>> from scipy import special
>>> special.hyperu(0.0001, 1.0, 0.0001)
-4806275004931.538
```

How can we determine the correctness of this result? Reading through some documentation and finding a function synonym of hyperu in version 1.0.0 of the mpmath library, we can attempt the same calculation, yielding a surprising discrepancy (albeit in mpmath's own mpf data type):

```
>>> import mpmath
>>> mpmath.hyperu(0.0001, 1.0, 0.0001)
mpf('1.0009210608660066')
```

Another comparison against the implementation of the Tricomi function found in version 2.6 of GSL returns a result that agrees with mpmath. Furthermore, SciPy's documentation for this hypergeometric function provides no information regarding the domain of the inputs that they do or do not handle. All of this serves to indicate a likely bug in the SciPy library.

Library users often find such bugs by chance. A previous study [19] showed that indeed, the above methodology is a common practice among users of numerical libraries to confirm or rule-out spurious results. However, manual differential testing on a case-by-case basis is not only time consuming, but it also assumes that the user has already discovered a problematic input, and that identifying equivalent functions in other libraries (and writing tests for them) is straightforward. An automated framework to uncover numerical bugs in the first place rather than confirming suspicions after the fact is the motivation for the work presented in this paper.

2.2 Definitions

Definition 2.1. Given a set F of functions, we define a binary relation $\triangleleft$ such that for any $f_1, f_2 \in F$, if $f_1 \triangleleft f_2$, then the set of analytical mathematical computations implemented by f_1 are a subset of those implemented by f_2.

Consider the generic functions, sqrt(x) and nthroot(n,x). We say that sqrt $\triangleleft$ nthroot. It then follows that all of the functionality of sqrt is differentially testable against nthroot, but not vice versa.

Definition 2.2. Given a set F of functions, we define an equivalence relation $\bowtie$ such that for any $f_1, f_2 \in F$, if $f_1 \bowtie f_2$, then $f_1 \triangleleft f_2$ and $f_2 \triangleleft f_1$. We then say that f_1 and f_2 are **function synonyms**.

Note that the above definition of function synonyms does not describe instances in which two functions can be made "equivalent" via the fixing of certain parameters to a specific value, e.g., sqrt(x) and nthroot(2,x), or the inlining of functions within larger expressions, e.g., euclidean_norm(x,y) and sqrt(pow2(x)+pow2(y)).

However, we do allow for a special case $f_1 \bowtie f_2$ *modulo data type* in which the set of possible arguments given to either f_1 or f_2 is restricted to an infinite[1] subset of itself. For instance, though gsl_bessel_In_scaled from GSL and ive from SciPy both implement the exponentially-scaled modified Bessel function of the first kind, the former requires that the first parameter be an integer whereas the latter allows for a double-precision value. Therefore, gsl_bessel_In_scaled $\triangleleft$ ive but not vice versa. However, restricting the first parameter of ive to the integer data type results in gsl_bessel_In_scaled $\bowtie$ ive *modulo data type*. We still refer to such cases as function synonyms.

Definition 2.3. Two function outputs y_1 and y_2 are considered **$\mathcal{E}$-equivalent** if they fall into one of three cases:

1) $y_1, y_2 \in \mathbb{R}$ and $|y_1 - y_2| < \mathcal{E}$.
2) $y_1, y_2 \in \{+\infty, -\infty, -0, \mathrm{NaN}\}$ and are the same.
3) y_1 and y_2 are both exceptions.

Definition 2.4. A **discrepancy** is a tuple (C, p, Φ, μ) in which C is the equivalence class of function synonyms, p is the input that provoked the discrepancy (i.e., the point of divergence), Φ is the set of outputs between which there exists at least one pair that is not $\mathcal{E}$-equivalent, and μ is a unique hash value identifying the discrepancy (see Section 3.3.2).

2.3 Scope of Our Approach

In this work, we focus on discovering discrepancies between *special functions*, defined to be "function[s] (usually named after an early investigator of its properties) having a particular use in mathematical physics or some other branch of mathematics" [45]. Furthermore, we target special functions that are made publicly available via each library's API and are meant to be used standalone by a client program. This choice of scope naturally fits differential testing for several reasons: firstly, special functions are widely implemented by numerical libraries and there are many of them. For instance, special functions constitute between 16% to 84% of all API functions in mpmath, SciPy, GSL, and jmat with an average of 50%. Secondly, the algorithms used to implement them are complex and diverse with different developers making different choices on a case-by-case basis.[2] Lastly, the arguments and return values of special functions are elementary data types, thus facilitating input generation and output comparison. For all of these reasons, such a focus has precedent in the literature [7, 20, 21, 41, 42, 48, 49].

For any libraries that do not clearly delineate in the source code what is and is not a special function (e.g., via namespaces or directory structures), we approximate by targeting functions in which each parameter and return value represents a real number. Consequently, functions that work exclusively with complex numbers, matrices, and arrays are not targeted by the approach described in this paper. Such a specification is made in an effort to maintain a balance between function complexity and ease of comparability so that differential testing is fruitful.

3 TECHNICAL APPROACH

Figure 1 depicts the three main components of FPDiff: (1) an extractor/generator that extracts signatures from library source code

[1]While the set of numbers representable with bits is not actually infinite, the term "infinite" here is used to refer to the set of numbers *represented* in memory.
[2]Interested readers are referred to *Numerical Methods for Special Functions* [23].

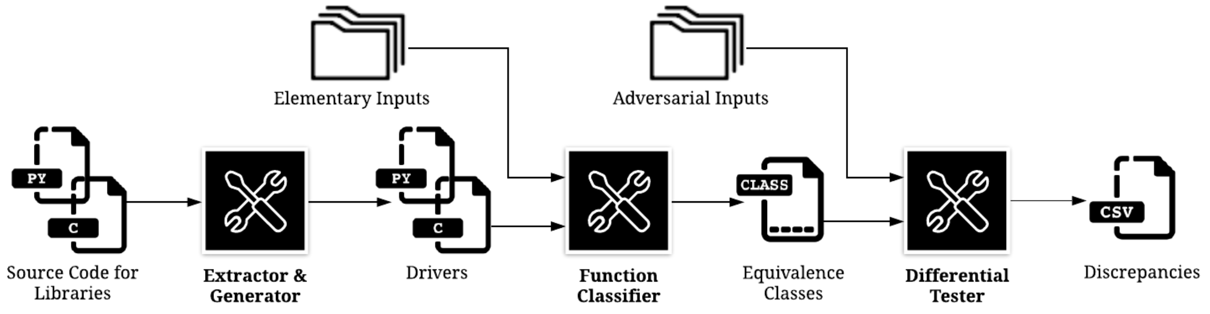

Figure 1: An overview of our framework which automatically collects function signatures, synthesizes executable drivers, discovers function synonyms, and applies differential testing to find points of divergence between numerical libraries.

and synthesizes executable drivers for numerical functions, (2) a classifier that clusters functions into equivalence classes of function synonyms based on their outputs when evaluated on *elementary inputs*, and (3) a differential tester that executes drivers within each equivalence class on a set of *adversarial inputs* and reports discrepancies found between function synonyms.

3.1 Drivers for Numerical Functions

3.1.1 Signature Extraction. Given a library's source code, the extractor returns a list of function signatures and the names of any potential imports/header files that might be required to call the functions. The strategy for extracting this information is dependent on each language's handling of types and, furthermore, on each library's conventions for declaring functions. In this paper, we implement FPDiff to target libraries written in both statically- and dynamically- typed languages.

Signatures when given type information. The inclusion of type information allows us to tailor the extractor to target only those functions that fit our scope, i.e., those with at least one double value in the input and that return a double value as the output. Once the library convention for defining functions is identified, a simple parsing of the source code for matching patterns suffices.

Signatures when given no type information. A potential source of type information is parsing the library's documentation, but the quality and consistency of that documentation becomes the limiting factor. For instance, `mpmath` makes no mention in the text of its documentation about the types of function parameters. Instead the user is expected to reason about types based on typeset mathematical formulas included in the documentation.

To address this challenge, we leverage developer-written tests to gather partial type information. To this end, we parse and traverse the AST for each test file to collect functions that, when called, have arguments that are either literals of type `int` or `double` or whose arguments are variables containing a reference to such values. In lieu of complete type information, all of these numerical arguments are collected and associated with the name of the called function. Additionally, we collect import statements found in these tests.

While this approach helps to filter out functions that do not take numerical arguments, it is limited: the literal arguments used to test the functions do not reliably indicate the type of the corresponding function parameter, e.g., when passing an integer 1 as an argument that the corresponding function synonym in a statically-typed language might require to be a double 1.0. Therefore, while the former extraction technique creates a collection of definitive function signatures with the desired parameter and return types, this technique yields a collection of function signatures that, when called in the library tests, had only numerical arguments. Such functions may have return values that fall outside of our scope. The technique for overcoming this ambiguity is addressed via instrumentation added to the drivers during their generation (detailed below) and the classification algorithm shown in Algorithm 1 and described in Section 3.2.

3.1.2 Driver Generation. Because FPDiff executes each function to identify function synonyms (Section 3.2) and to discover points of divergence (Section 3.3.2), the goal of the driver generator is to automatically create a harness around each function that facilitates the function evaluation.

The driver generator receives input from the extractor consisting of a collection of function signatures (with or without complete type information) and any necessary header files or imports. The generator inserts this information into a template to create drivers like the `GSL` and `SciPy` drivers shown in Figure 2. Each driver must meet the following criteria:

(1) Feed arguments from a uniform set of inputs to facilitate both the placement of the functions into equivalence classes based on their evaluation over elementary inputs and the subsequent differential testing over the set of adversarial inputs. This is accomplished by parameterizing each driver with two arrays of double-precision floats and integers (first lines of Figures 2a to 2c) such that a function with n parameters will involve n inputs read from the appropriate arrays (highlighted lines in Figures 2a to 2c).
(2) Allow for exceptions within the library code to propagate upward to the calling procedure to be collected. For example, Line 4 in Figure 2a accomplishes this by overriding the default error handler for `GSL`, which aborts program execution.
(3) Provide a means of raising an appropriate exception when attempting to execute a driver for a function which, though found

```
1 double gsl_sf_bessel_In_scaled_DRIVER(char*
      doubleInput, char* intInput) {
2   double out;
3
4   gsl_error_handler_t * old_handler =
        gsl_set_error_handler (&my_handler);
5
6   out = gsl_sf_bessel_In_scaled(intInput[0],
7   doubleInput [0]);
8   return out;
9 }
```

(a) Definitive driver synthesized for the GSL function `gsl_sf_bessel_In_scaled`.

```
1 def scipy_special_ive_DRIVER0(doubleInput, intInput):
2    out = special.ive(doubleInput[0], doubleInput[1])
3    return float(out)
```

(b) One of the two drivers synthesized for the `SciPy` function `ive`. Though executable, this driver will not be mapped to the analogous `GSL` function shown in (a) because `ive` will not be fed the same inputs.

```
1 def scipy_special_ive_DRIVER1(doubleInput, intInput):
2    out = special.ive(intInput[0], doubleInput[0])
3    return float(out)
```

(c) Second driver for the `SciPy` function `ive`. This driver will successfully be mapped to the analogous `GSL` function shown in (a).

Figure 2: Examples of Generated Drivers

by the signature extractor, does not fit our chosen scope, e.g., a function that was present in the developer tests with double inputs but returns arrays. Line 3 in Figures 2b and 2c accomplishes this by raising an exception when the return of the function call is not a floating-point number.

(4) For those functions for which we do not have complete type information, the driver must utilize the proper configuration of integer and double arguments to be correctly mapped to potential function synonyms. See discussion below.

Fulfillment of the fourth criteria allows our pipeline to automatically perform differential testing between libraries that do not provide type information. Consider the example from Section 2.2 `gsl_sf_bessel_In_scaled` $\bowtie$ `ive` *mod data type*.

Because the signature for `gsl_sf_bessel_In_scaled` was extracted from a library with type information, the generator synthesizes the single definitive driver shown in Figure 2a in which the first parameter is an integer and the second is a double. However, even though `ive` from `SciPy` is the proper function synonym, its signature was extracted without type information. Thus, `ive` will only appear functionally identical to `gsl_sf_bessel_In_scaled` if given the same argument types. In other words, we have to ensure that the first argument to `ive` comes from the array of integer inputs just like `gsl_sf_bessel_In_scaled`.

In order to address this challenge, the driver generator will create multiple drivers for each extracted function signature that lacks complete type information, each with a different configuration of integer and double data types. Figures 2b and 2c illustrate the two drivers synthesized for the `SciPy` function `ive` with their difference highlighted. During the classification process discussed in Section 3.2, only the drivers with matching data type configurations (in this case, Figure 2a and Figure 2c) will be recognized by the classifier as function synonyms.

3.2 Function Classification

Our classifier addresses one of the main challenges of differential testing: identifying what function pairs are "equivalent" and therefore acceptable for such testing. A set of drivers is supplied as input to the classifier which outputs a set of equivalence classes that indicate the existence of likely function synonyms (Definition 2.2).

Algorithm 1 describes our methodology. First, each function is assigned a *characteristic vector* of numerical values that aims to represent the underlying semantics of the function (Lines 3 through 8). Equivalence classes are then constructed based on the similarity of these vectors (Lines 11 through 19). To generate each characteristic vector, we leverage a property of any relatively well-written numerical algorithm: for the overwhelming majority of its input domain, the function is well-behaved [37]. Note that because the space of possible floating-point values is so vast, each output inherently encodes detailed information about the underlying calculation used to arrive at that value: a collection of such outputs therefore captures a portion of the semantics of the program that generated it in a form that is easily comparable via standard arithmetic operations. In contrast, a numerical function that returns integer values has outputs that contain much less information about the program's underlying semantics by way of the pigeonhole principle.[3]

To this end, our classifier evaluates each synthesized driver over a set of *elementary inputs*. Elementary inputs are values for which most functions are likely to exhibit well-defined behavior; the resulting set of outputs constitutes the characteristic vector. Note that the else branch on Line 7 of Algorithm 1 handles the possibility that an evaluation over an elementary input does not exhibit well-defined behavior by logging an output of `NaN`. Furthermore, the conditional statement on Line 9 removes unwanted drivers that exhibited undesirable behavior on every evaluation over every elementary input and, therefore, have a characteristic vector of all `NaN` values; this works in tandem with the driver instrumentation required by the third criteria for successful drivers as stated in Section 3.1.2.

For the classification based on characteristic vectors, the conditional on Line 13 ensures that only functions with the same number and types of parameters are compared. Finally, the conditional on Line 14 implies an element-wise comparison of the elements in each characteristic vectors for $\mathcal{E}$-equivalence (see Definition 2.3). Note that the size of the $\mathcal{E}$-neighborhood varies depending on the size of the values being compared; intuitively, larger values will have larger $\mathcal{E}$'s. To address this, FPDiff users define a relative tolerance ϵ such that if the relative error between any two values is less than ϵ, the values will be considered $\mathcal{E}$-equivalent.

3.3 Differential Testing

Our differential tester takes a set of equivalence classes and evaluates drivers over a set of *adversarial inputs*, i.e., inputs that are

[3] https://en.wikipedia.org/wiki/Pigeonhole_principle

Algorithm 1: Function Classification

```
Input: Drivers, elementaryInputs
Output: equivalenceClasses
equivalenceClasses = []
for driver in Drivers do
    for doubleInput, intInput in elementaryInputs do
        result = driver(doubleInput, intInput)
        if isNumerical(result) then
            driver.characteristicVec.append(result)
        else
            driver.characteristicVec.append(NaN)
    if allNaN(driver.charateristicVec) then
        continue
    match = False
    for class in equivalenceClasses do
        if sameParameterConfig(class[0],driver) then
            if sameCharacteristicVec(class[0],driver) then
                match = True
                class.append(driver)
                break
    if not match then
        equivalenceClasses.append([driver])
return equivalenceClasses
```

intended to provoke divergences between function synonyms. Outputs are then compared for $\mathcal{E}$-*equivalence* with points of divergence reported as a *discrepancy* (see Definitions 2.3 and 2.4).

There are three remaining challenges to enable differential testing of numerical functions: (1) How do we obtain effective adversarial inputs? (2) How do we triage discrepancies found? (3) How do we automatically reduce the set of reported discrepancies to facilitate manual inspection? We discuss these challenges below.

3.3.1 *Sources for Adversarial Inputs.*

The set of inputs for which a function produces significant divergences is infinitesimal compared to the size of the possible domain [48, 49]. It is this fact that both motivates our technique for classifying functions (Section 3.2) and represents one of the major challenges for differential testing of numerical libraries. Discovering such inputs is a well-studied problem [5, 9, 15, 16, 21, 25, 49]; though we choose three sources of inputs in this paper, it is worth noting that other input sources may be used. For this work, our adversarial inputs come from tests written by developers, a guided binary-search of a portion of the domain, and special values as defined by the IEEE 754 Standard.

Test Migration Inputs. With test migration, we leverage the assumption that inputs used in test programs are carefully chosen by developers to expose corner cases in the algorithms being tested. During the signature extraction process described in Section 3.1.1, we collect such inputs and associate them with the function signature they were used with in order to migrate them to function synonyms within the same equivalence class.

Inaccuracy-Inducing Inputs. For each pair of function synonyms and for a specified domain range, we perform a binary guided random search for inputs that maximize the relative error between the two functions.[4] These inputs are then collected and applied to the rest of the functions in the equivalence class. The assumption here is that inputs found to cause inaccuracies between one pair of function synonyms will have a higher likelihood of prompting divergence between other functions within the equivalence class.

Special-Value Inputs. In addition to the finite set of floating-point numbers representable within a particular format, the IEEE 754 Standard also defines a set of *special values*: negative zero, positive infinity, negative infinity, and `NaN`.[5] These values can arise as the result of certain floating-point operations and, as such, give rise to the possibility that they might be used as input to another numerical function; therefore, while not strictly a part of an algorithm to implement a mathematical function, it is still the responsibility of developers to handle such inputs. Moreover, documentation for libraries is often lacking when it comes to describing behavior for inputs outside the expected domain if it mentions domain at all. Using these values as adversarial inputs in our differential testing will illustrate the choice (or lack thereof) that different developers make on how to handle special values.

3.3.2 *Discrepancy Triage.*

Recall that each discrepancy is defined as a tuple (C, p, Φ, μ) representing respectively the equivalence class of function synonyms, the point of divergence, the outputs exhibiting the divergence, and a unique identifying hash value (see Definition 2.4).

Our analysis first maps each element of p to an element of the abstract domain consisting of `NaN`, `infinite`, `zero`, `negative`, and `positive`. The result is the abstract input $\hat{p}$. For example, if p = `{0, -1.1, -inf}`, then $\hat{p}$ = `{zero, negative, infinite}`. Each $\hat{p}$ is then used to calculate μ, described in Section 3.3.3, and categorize the discrepancy.

The discrepancy categories are summarized in Table 1 and are loosely ordered by their likelihood of representing buggy behavior. Such a categorization is intended to group together discrepancies that likely indicate similar undesirable behaviors in one or more libraries and serves as one of our key contributions that enable FPDiff to effectively report bugs. These categories are a result of a manual inspection of discrepancies discovered via our testing framework. Each category is described and discussed below.

Category 6 discrepancies are those in which a hang is observed in at least one of the libraries. Such discrepancies clearly represent buggy behavior; programs should terminate gracefully regardless of input, making a hang unacceptable.

Category 5 discrepancies are those in which a double output is generated in at least one of the libraries from a $\hat{p}$ containing a `NaN` *value.* A `NaN` input could be the return value of another function indicating that some illegal operation took place (e.g., zero divided

[4]Chiang et al. [16] find inputs that maximize error between a program and its high-precision version, and focus on C programs. We adapt their technique to maximize error across distinct implementations of numerical functions, written in Python and C.
[5]Though the IEEE 754 Standard defines a quiet `NaN` and a signalling `NaN`, the languages we focus on do not define the behavior of the latter and use `NaN` to refer to the former.

Table 1: A categorization of discrepancies, loosely ordered by likelihood of representing buggy behavior.

Category	Discrepancy Description
6	A hang is observed in at least one of the libraries.
5	Double output is generated in at least one of the libraries from a $\hat{p}$ containing a `NaN` value.
4	Mix of doubles, exceptions, and/or special values from a $\hat{p}$ containing an `infinite` value.
3	A relative error greater than a chosen ϵ is detected between two libraries.
2	Mix of doubles, exceptions, and/or special values generated from a $\hat{p}$ containing only `positive`, `negative`, or `zero` values.
1	Mix of special values and exceptions generated from any input type.

by zero, logarithm of a negative number, etc.). If the function receiving that `NaN` value produces a double value as result, then the fact that an illegal operation occurred prior to the function call is lost. In such cases, it is also troubling to consider the fact that an input representing "not-a-number" somehow gave rise to a number as an output. Category 5 discrepancies have a high potential for representing true bugs.

Category 4 discrepancies are those in which a mix of doubles, exceptions, and/or special values are generated from a $\hat{p}$ containing an `infinite` *value.* The existence of such discrepancies may be due to differing design choices between libraries on whether or not to support the evaluation of limits approaching infinity. Domain-specific knowledge regarding the asymptotic behaviors of the special function in question is required to determine whether or not each discrepancy represents a true bug.

Category 3 discrepancies are those in which a relative error greater than a chosen ϵ is detected between two libraries. This indicates that the libraries cannot agree on the result of a legitimate computation; the seriousness of such an error has been well-documented with tragedies such as the Patriot Missile failure and the explosion of the Ariane 5 rocket [3, 4].

Category 2 discrepancies are those in which a mix of doubles, exceptions, and/or special values are generated from a $\hat{p}$ containing only `positive`*,* `negative`*, or* `zero` *values.* These discrepancies suggest that the libraries do not agree on whether or not a problem occurred in the computation. Such ambiguity is obviously undesirable, though diagnosing this behavior as buggy can require substantially more manual effort than the other categories above; see Section 4.3 for a discussion of some specific examples.

Category 1 discrepancies are those in which a mix of special values and exceptions are generated from any input type. These discrepancies are distinct from those in categories 2, 4, and 5 because no double outputs are observed. This indicates that, while the libraries all appear to agree on the fact that a problem occurred in the computation, they disagree on how to handle such a problem. Category 1 discrepancies illustrate the lack of unifying standards for error handling numerical libraries, and do not necessarily constitute reportable bugs.

While some of these categories are potentially discoverable by systematically executing drivers over different inputs, doing so within equivalence classes of function synonyms gives users the added benefit of either suggesting the correct behavior (when there is a general consensus between the other function synonyms) or demonstrating the variety of different choices made by developers in handling adversarial inputs.

As an additional measure to aid in discrepancy triage, FPDiff also utilizes a δ-difference metric as described by Klinger et al. [29] that quantifies for each function how many other function synonyms within the equivalence class have outputs that disagree with that function. For example, if function f_1 returns `NaN` and its function synonyms f_2, f_3, and f_4 all return 0 for a specific adversarial input, the δ-differences will be 3 for f_1 and 1 for the others. FPDiff tags each discrepancy with the maximum δ-difference between all the function synonyms. Larger values will therefore correspond to discrepancies present in larger equivalence classes in which a minority of function synonyms disagree with the majority.

3.3.3 Reducing Reported Discrepancies. In order to minimize the reporting of multiple discrepancies that are all representative of the same buggy behavior, we ensure that the each discrepancy output by the differential tester is assigned a μ value that encodes the defining characteristics of each discrepancy: the equivalence class of function synonyms in which the discrepancy was observed, the point of divergence, and the set of results that disagree. With this in mind, μ is calculated as the hash value of the input tuple consisting of the summation of the equivalence class' characteristic array, the abstracted input tuple $\hat{p}$, and the category of the discrepancy. In the case of multiple category 3 discrepancies with the same μ, the one with the largest relative error is reported. In all other cases, the first discrepancy found is the one reported.

Additionally, all category 1 discrepancies are removed from the final reduced log of discrepancies. As discussed above, such discrepancies indicate the different techniques library developers use to handle inevitable numerical errors. While their existence is informative in its own right, they do not represent reportable bugs.

4 EXPERIMENTAL EVALUATION

This evaluation is designed to answer the following questions:

RQ1 How effective are our signature extractor and driver generator at discovering functions and synthesizing their drivers?

RQ2 How effective is our classifier at discovering function synonyms within and across numerical libraries?

RQ3 What sorts of discrepancies does FPDiff discover between function synonyms?

RQ4 Does differential testing discover discrepancies that represent reportable bugs? If so, how difficult is it to ascertain which discrepancies are reportable?

Selection of Numerical Libraries. We consider four numerical libraries: the GNU Scientific Library (GSL), SciPy, mpmath, and jmat. We chose open-source numerical libraries that have overlapping functionality, are widely-used in practice, and cover a set of very different programming languages. GSL is a C library of numerical routines whose special functions are a popular subject of experimental evaluation for numerical testing like this (e.g., [7, 21, 22, 41, 42, 48, 49]). SciPy is a well-known Python numerical library that implements a variety of numerical routines and is used by more than 136,000 GitHub repositories. mpmath is a Python numerical library that implements much of the functionality of SciPy with additional support for arbitrary-precision arithmetic and is a standard package included in both the SymPy and Sage computer algebra systems. jmat is a JavaScript numerical library that provides implementation for special functions, linear algebra, and more; it is the most popular JavaScript library with this functionality found on GitHub.

Experimental Setup. The classifier uses a relative tolerance value of $\epsilon = 10^{-8}$ (on the order of machine epsilon) to determine $\mathcal{E}$-equivalence. FPDiff uses a relative tolerance value of $\epsilon = 10^{-3}$ to determine what amount of inaccuracy is considered severe enough to report as a category 3 discrepancy (previously, we used the same relative tolerance as the classifier but initial feedback showed that some developers considered such inaccuracies too small to be significant, even if their library was the only outlier). In the absence of a ground truth by which to scale the absolute error, we use the magnitude of the relative percent difference, shown in equation 1, as our relative error. Any references to relative error within our evaluation are taken to mean the relative percent difference.

$$\frac{|x_1 - x_2|}{(|x_1| + |x_2|)/2} = 2\frac{|x_1 - x_2|}{|x_1| + |x_2|} \tag{1}$$

The classifier executes each driver over 40 elementary inputs to yield characteristic vectors. Elementary inputs of double type are randomly and uniformly generated between 0 and 3 exclusive and integer inputs are either 0, 1, 2, or 3. The choice of elementary inputs was guided by intuition and refined heuristically. Roughly speaking, most special functions tend to exhibit their most interesting/defining behavior around zero and are more likely to be defined for positive values than negative values.

Hangs constituting category 6 discrepancies are discovered by setting a timeout threshold for each function's execution. For our experiments, we chose a time limit of 20 seconds.

We generated a collection of inaccuracy-inducing inputs using a modified version of the s3fp tool developed by Chiang et al. [15] which uses binary guided random testing to find inputs that maximize the floating-point error between 128-bit precision and 64-bit precision versions of a single C program. We modified their source code to also work with Python programs, to use 64-bit double data types, and to calculate relative errors with equation 1. For each pair of function synonyms, we allowed an hour of search time within the domain interval $[-100, 100]$ as done in their original evaluation.

Each function in the mpmath library is present in two different namespaces: one that corresponds to an arbitrary-precision implementation and one for a floating-point-precision implementation. Using this knowledge, the driver generator gives us a pair of drivers for each mpmath function. In the following evaluation, we do not count these pairs as function synonyms discovered by FPDiff, we do not double count mappings to each mpmath function, and we do not count equivalence classes that only contain these pairs.

FPDiff is implemented as a collection of Python modules which can be found at https://github.com/ucd-plse/FPDiff. The documentation includes directions for those who wish to extend FPDiff to add new libraries for testing. We ran our experiments on a workstation with a 3.60GHz Intel i7-4790 and 32 GB of RAM running Ubuntu 18.04.

Table 2: Evaluation of the Driver Generator

Library	Eligible Functions	# Captured	% Captured
GSL	193	193	100%
jmat	154	154	100%
mpmath	211	147	69.7%
SciPy	206	180	87.4%
Total	764	674	88.2%

4.1 Effectiveness of Extractor and Driver Generator

To evaluate the effectiveness of the signature extractor and driver generator, we report the recall with respect to drivers synthesized by FPDiff for functions eligible for our experimental evaluation. We omit precision calculations because, as discussed in Section 3.1.2, the extractor/generator component creates drivers for a superset of the targeted functions (in the case of libraries without type information) which is handled with a combination of driver instrumentation and the function classification described by Algorithm 1. Also note that these components work in tandem and, as such, are evaluated together; if the extractor successfully captured a function signature, the generator will synthesize a driver for that signature.

We manually examine the documentation for the chosen libraries to construct the ground truth of eligible functions for our study (see Section 2.3 for the scope). For SciPy and mpmath's functions that include optional parameters with default arguments, we count each possible configuration as a unique function signature. The results are shown in Table 2.

Overall, we successfully generate an executable driver for 674 out of 764 eligible functions (88.2%). Note that the 100% recall for GSL and jmat is unsurprising due to the fact that both libraries include type information in their signatures, allowing us to simply parse source code for matching patterns.

We found the success of these components to be highly correlated with the thoroughness of library test code. For example, of the 64 mpmath functions we did not capture, 57 of them were not present at all in the developer tests. The remaining 7 functions were not captured because they were tested with named constants defined elsewhere in the library or arguments to lambda expressions; in these cases, our extractor was not able to determine that these function's parameters were numerical arguments as required by our experiment's scope.

Table 3: Function Synonym Mappings Precision and Recall

Library Pair	Precision	Recall
`GSL/jmat`	27/27 (100%)	27/38 (71.1%)
`GSL/mpmath`	61/65 (93.8%)	57/69 (82.6%)
`GSL/SciPy`	68/69 (98.6%)	67/76 (88.2%)
`jmat/SciPy`	29/30 (96.7%)	27/43 (62.8%)
`mpmath/jmat`	44/45 (97.8%)	40/49 (81.6%)
`mpmath/SciPy`	47/48 (97.9%)	44/56 (78.6%)
same library	21/26 (80.8%)	16/21 (76.2%)
Total	297/310 (95.8%)	278/352 (79.0%)

While all but two of the eligible `SciPy` functions appeared in the developer written tests, 24 of the 26 functions that did not receive executable drivers were missed by the extractor because they were either passed as function objects without parameters to a separate testing function or their arguments were `numpy.ndarray` objects.

RQ1: It is not difficult to obtain a 100% capture rate for libraries containing type information; for those without such information, the success of leveraging developer tests is highly dependent on the quality of those tests. More mature libraries such as `SciPy` are more likely to have more exhaustive tests which benefits our technique.

4.2 Effectiveness of Function Classifier

To establish the ground truth for the set of mappings FPDiff should capture, we manually placed all of the eligible functions into equivalence classes based on documentation, source code, and other external resources. Our function classifier reported 310 pairs of function synonyms with 95.8% precision and 79.0% recall. A breakdown per-library is shown in Table 3. The set of correct function synonyms constituted 126 equivalence classes, encompassing 498 functions ready for differential testing.

Precision was calculated by manually verifying each mapping reported by FPDiff. The source of imprecision in all cases was the choice of elementary inputs; while equivalent for positive inputs, these false function synonyms diverge when the inputs are negative. It is worth noting that while there were 13 incorrect mappings, these were the consequence of only 7 misplaced functions.

To demonstrate the effectiveness of our classifier technique, consider the set of Bessel functions. Between the four libraries, there are about 90 such functions with each library using their own naming conventions to distinguish between them. FPDiff discovers 28 equivalence classes of Bessel functions, one of which encompasses the regular cylindrical Bessel functions of the first kind with integer order, shown below:

$$\begin{cases} \texttt{gsl\_sf\_bessel\_Jn} & (\texttt{GSL}) \\ \texttt{angerj} & (\texttt{jmat}) \\ \texttt{besselj} & (\texttt{mpmath}) \\ \texttt{jn} & (\texttt{SciPy}) \\ \texttt{jv} & (\texttt{SciPy}) \\ \texttt{jve} & (\texttt{SciPy}) \end{cases}$$

First, such mappings are non-obvious based on names alone. For instance, `gsl_sf_bessel_jl` and `gsl_sf_bessel_Jnu` are correctly determined to *not* belong to this equivalence class. Second, we see a valid intra-library function synonym, `jn` $\bowtie$ `jv`, despite the fact that `jn` does not exist in the `SciPy` documentation. Third, we observe a number of valid function synonyms *mod data type* (see Section 2.2): `gsl_sf_bessel_Jn` is the implementation of the regular cylindrical Bessel function of the first kind with *integer* order. `jn`, `jv`, and `besselj` implement the same function, but for any *fractional* order. `jve` is the same as `jn` and `jv`, but scaled by an exponential of the imaginary part of the input. `angerj` is a generalization of the Bessel function of the first kind. All of these become synonyms mod data type when their first parameter is an integer.

As for recall, the main causes for missing mappings were a failure in the extractor to get the function in the first place or inaccuracies in a function's evaluation over elementary inputs that exceeded the choice of ϵ. For instance, while FPDiff reported the six functions above for the equivalence class of cylindrical Bessel functions of the first kind with integer order, it failed to discover a seventh, the `jmat` function `besselj`. This is because inaccuracies in the function caused several of the elements in its characteristic vector to fall outside of the acceptable $\mathcal{E}$-neighborhood. However, FPDiff did correctly place `jmat`'s `besselj` in the equivalence class representing the regular cylindrical Bessel functions of the first kind with *fractional* order mentioned above.

Note that in the set of reported equivalence classes, there existed 5 correctly mapped functions that were not in the set of eligible functions that we manually constructed. This was because we used the documentation as reference when constructing the ground truth and these 5 functions were not listed in the documentation (as in the case of `jn` above). Any mappings to these functions were therefore not counted in the recall portion since they were not in the ground truth set.

RQ2: FPDiff finds function synonyms with a 95.8% precision and 79.0% recall, thus demonstrating the effectiveness of classifying functions based on their evaluation over elementary inputs. This approach even managed to correctly place functions that were not present in the documentation. However, efficacy is influenced by the choice of elementary inputs. Improvements in the extractor/generator component would benefit recall.

4.3 Discrepancies Found

After removing discrepancies with identical values of μ, FPDiff found a grand total of 655 unique discrepancies between function synonyms from the `GSL`, `jmat`, `mpmath`, and `SciPy` numerical libraries. For the manual inspection for bugs that followed, all 328 category 1 discrepancies were removed to give a reduced set of 327. Of these, we found 150 of them to represent 125 unique bugs. Specifically, FPDiff identified 31 bugs in `GSL`, 33 bugs in `jmat`, 44 bugs in `mpmath`, and 17 bugs in `SciPy`. We have reported all bugs to library developers; so far, 30 bugs have been fixed, 9 have been found to be previously known, and 25 more have been acknowledged by developers.

Table 4 shows the counts of unique discrepancies found by FPDiff broken down by adversarial input source and discrepancy

Table 4: Discrepancies found per input source and category.

	Category #						
	6	5	4	3	2	1	Total
Special Values	20	23	95	7	11	280	436
Test Migration	-	-	-	43	107	48	198
Inaccuracy-Inducing	-	-	-	13	12	1	26
Overall (Unique)	20	23	94	61	129	328	655
Reportable	**19**	**21**	**37**	**23**	**50**	**0**	**150**

Table 5: Bugs found per library and discrepancy category.

	Category #						
	6	5	4	3	2	1	Total
GSL	-	8	12	2	9	-	31
jmat	-	5	8	8	12	-	33
mpmath	17	7	10	1	9	-	44
SciPy	-	7	2	2	6	-	17
Total	17	27	32	13	36	-	125

category (see Table 1 for category descriptions), as well as counts of discrepancies containing reportable buggy behavior and unique bugs found per discrepancy category. In total, FPDiff used 160 special-value inputs, 4,513 migrated test inputs, and 1,092 inaccuracy inducing inputs. Each test-migration and inaccuracy-inducing input was applied only to the equivalence class from which it was found while each of the special value inputs was applied to *all* equivalence classes. This combined with the fact that special value inputs were the only adversarial inputs to yield category 4, 5, and 6 discrepancies explains why the relatively smaller number of unique inputs generated so many more discrepancies. Migrated test inputs were significantly more effective than inputs discovered via a binary-guided random search, generating over 7 times as many overall discrepancies and over 3 times as many inaccuracies; this lends support to our hypothesis that inputs used in test programs are carefully chosen by developers to expose corner cases in the algorithms being tested.

Table 5 shows the break down of reportable bugs per library and discrepancy category. FPDiff helped us to identify the most bugs in mpmath, followed by jmat, GSL, and SciPy. With the exception of category 6, the distribution of bugs across libraries was mostly even (with a couple of outliers in categories 3 and 4, see discussion below). This demonstrates that the discrepancies found by FPDiff and the bugs that they represent are not specific to a single numerical library. Note that categories 2 and 4 represented the highest number of bugs (36 and 32, respectively) while category 3 represented the fewest (13 bugs).

We now offer a series of case-studies surrounding examples discovered by FPDiff from each discrepancy category.

4.3.1 Category 6 Discrepancies. Because hangs are certainly buggy behavior, every category 6 discrepancy has a high probability of representing a reportable bug. Points of divergence that caused hangs all included NaN or infinite values, leading to a natural suggestion for a fix: checking inputs for special values. Such hangs were only observed in mpmath with only 1 of the 20 not being reproducible (this instance was due to an exception being raised regarding lack of convergence when the function's execution was allowed to exceed our chosen cutoff). While these hangs were acknowledged by mpmath developers, fixes have yet to be applied, which might indicate that the apparent simplicity of the fix may be misleading (see discussion below).

4.3.2 Category 5 Discrepancies. Consider the following discrepancy discovered by FPDiff:

```
(jmat)        factorial(NaN) ⇒ NaN
(mpmath) mp.factorial(NaN) ⇒ NaN
(mpmath) fp.factorial(NaN) ⇒ EXCEPTION
(SciPy)       factorial(NaN) ⇒ 0
```

The majority behavior of propagating the NaN, as observed in jmat and the arbitrary-precision mpmath implementations, suggests a more appropriate action than simply returning 0. After filing a report, this was confirmed in the ensuing discussion with SciPy developers, who stated, "inputs in the array that enter as NaN should remain NaN." Thus, our bug report ultimately led to a fix to be included in the 1.5.0 release of SciPy.

Note that even though such fixes may be conceptually simple (check input for NaN, return a NaN if found, otherwise return the function result), the complex structure of numerical software makes for a non-trivial implementation. For this particular example with scipy.special.factorial, the pull request (PR) containing only this bug fix included 8 commits, 26 changes, and a back-and-forth between maintainers encompassing dozens of comments that lasted almost a month. Interested readers are encouraged to examine the thread accompanying the PR.[6]

It is also worth noting the discrepancy between the arbitrary-precision and floating-point-precision functions in mpmath. The floating-point implementation throws an exception with the message "cannot convert float NaN to an integer" while the arbitrary-precision implementation propagates the NaN value. It is not difficult to contrive a scenario in which a user swaps the two implementations within some larger program, assuming they are equivalent, thus possibly giving rise to unexpected failures.

4.3.3 Category 4 Discrepancies. Consider the following discrepancy discovered by FPDiff between functions implementing the *Dirichlet eta function*, also known as the *alternating zeta function*:

```
(GSL)    gsl_sf_eta(inf) ⇒ 1
(jmat)          eta(inf) ⇒ NaN
(mpmath) mp.altzeta(inf) ⇒ 1
(mpmath) fp.altzeta(inf) ⇒ 1
```

Again, the majority-rules intuition suggests that this function asymptotically approaches 1. A confirmation of this via an external resource and the fact that the jmat library supports evaluation of limits led to a bug report and a fix.

Note from Table 5 that only 2 out of 32 bugs related to category 4 discrepancies were were attributed to SciPy. This might indicate

[6]https://github.com/scipy/scipy/pull/11254

that `SciPy` has a more consistent policy for the evaluation of limits in special functions than the other libraries.

4.3.4 Category 3 Discrepancies. The example shown in Section 2.1 is a category 3 discrepancy that was discovered by FPDiff using a migrated test input from `GSL`. Searching the issue tracking system for `SciPy` showed similar reports dating back to 2013 regarding such inaccuracies in the *Tricomi confluent hypergeometric function.* A recent comment from a developer on an issue filed in late 2019 stated, "`hyperu` definitely needs more improvements. We're getting there, but it's a process." This demonstrates the fact that, unlike those in categories 4-6, discrepancies in category 3 represent bugs that do not suggest a natural or easy fix.

The fact that only a single inaccuracy bug was discovered in `mpmath` is unsurprising; `mpmath` implements arbitrary-precision arithmetic and should therefore be less susceptible to such defects. On the other hand, more than half of the bugs from this discrepancy category (8 out of 13) were attributed to `jmat`. This might be a consequence of the relative immaturity of the software with respect to the others it was tested against (the first commit to `jmat` repository was in 2014) and its smaller group of contributors.

4.3.5 Category 2 Discrepancies. The following example shows two discrepancies for the same equivalence class containing function synonyms implementing the *exponential integral* or *En-function*:

```
(GSL)     gsl_sf_expint_En(-2,-2) ⇒ EXCEPTION
(mpmath)            expint(-2,-2) ⇒ -1.84...
(SciPy)               expn(-2,-2) ⇒ inf

(GSL)     gsl_sf_expint_En(1,-1) ⇒ -1.89...
(mpmath)            expint(1,-1) ⇒ EXCEPTION
(SciPy)               expn(1,-1) ⇒ inf
```

The exception thrown by `GSL` came with the message `domain error` while the exception thrown by `mpmath` actually came from the driver around the `expint` function that complained when the return value was a complex number. Coming to conclusions about this particular set of category 2 discrepancies requires some domain-specific knowledge and a little experimentation.

The *En-function* has a branch cut along the negative real axis so for that portion of its domain, these developers made different choices about which value to return. When consulting the documentation, `SciPy` is the only library that provides an expected domain, stating that both arguments should be greater than or equal to zero. Furthermore, in the examples given in the documentation, they demonstrate that the expected return for inputs outside of the domain is a `NaN`. As a result, though the differing values are not necessarily buggy, reports were filed for the lack of documentation for `GSL` with respect to an expected domain, and the inconsistency with which `SciPy` handles unexpected inputs.

4.3.6 Category 1 Discrepancies. Discrepancies in this category are unlikely to represent reportable bugs, though they do illustrate the ways in which intentional choices made by developers can create points of divergence. Consider the following example of a category 1 discrepancy discovered by FPDiff in the equivalence class of functions implementing the Bessel function of the second kind:

```
(GSL)     gsl_sf_bessel_Yn(0,-0.5) ⇒ EXCEPTION
(jmat)            bessely(0,-0.5) ⇒ EXCEPTION
(mpmath)          bessely(0,-0.5) ⇒ EXCEPTION
(SciPy)                yn(0,-0.5) ⇒ NaN
```

The exceptions from `jmat` and `mpmath` are from their generated drivers complaining about complex returns and the `GSL` exception is a `domain error`. From this, we observe that while `jmat` and `mpmath` provide support for complex numbers, `GSL` and `SciPy` do not. Furthermore, the means by which `GSL` and `SciPy` developers choose to handle such computations outside of their chosen scope differs, i.e., throwing exceptions versus generating `NaN` values.

RQ3: FPDiff discovered 655 unique discrepancies between function synonyms encompassing all six categories of numerical discrepancies. Different sources of adversarial inputs vary in their effectiveness and the type of discrepancies they reveal. Special-value inputs were particularly effective and test-migration inputs proved to be superior to those gathered via a binary guided random search.

RQ4: Ignoring category 1 discrepancies, we found that about 46% (150/327) of discrepancies concerned reportably buggy behavior. These 150 discrepancies represented 125 unique bugs. While identifying such discrepancies can be labor intensive and sometimes requires domain-specific knowledge, those of higher category or larger maximum δ-difference values are more easily diagnosed as buggy.

4.4 Threats to Validity

In this paper, we implement systematic and automated differential testing to discover discrepancies across numerical libraries. However, our approach has a number of limitations. *First*, our approach only targets a subset of the functions present in numerical libraries. Even so, special functions make up a large class of complex functions that are widely-studied in the literature. The results of our experiments stand on their own. *Second*, our extractor strategy for libraries with incomplete type information is limited by the thoroughness of developer-written tests. We aimed to reduce this threat by including popular numerical libraries that are actively used and developed. *Third*, the choice of elementary inputs can cause the classifier to miss some mappings. We attempted to increase our recall by running the pipeline multiple times and refining the range of inputs heuristically, to positive results. *Fourth*, not all discrepancies represent buggy behavior; results must be manually inspected to determine whether or not a discrepancy is a bug. We take strides to remedy this by proposing a categorization that can be used to effectively triage discrepancies; by automatically removing category 1 discrepancies, we reduce the number to be inspected by half.

5 RELATED WORK

Numerical Software Testing. A variety of tools have been developed for testing numerical software, from which a large number deal with input generation. Fu and Su [21] and Bagnara et al. [5] generate inputs that maximize code coverage via unconstrained programming and symbolic execution, respectively. Others maximize the floating-point error in the result via genetic algorithms [49], binary search over a specified domain [16], and symbolic execution [25]. In addition to maximizing error, recent work [48] also applies a patch of piecewise quadratic functions to approximate the desired behavior. Barr et al. [7] leverage symbolic execution to discover inputs that trigger exceptions. Chiang et al. [15] discover diverging results in certain types of numerical programs. All the above techniques are complementary to our approach.

To address the missing oracle problem, Chen et al. [12] leverage invariant properties referred to as "metamorphic relations". This technique has seen success in testing certain numerical routines. For instance, LAPACK [2] runs compile-time tests for their Linear Equation and Eigensystem routines using such metamorphic relations. However, such relations require domain-specific knowledge and cannot possibly cover all corner cases. Kanewala and Bieman [28] apply machine learning for mining metamorphic relations.

Other approaches focus on analysis of numerical code. Bao and Zhang [6] use a bit tag to track the propagation of inaccuracies through a program's execution. Benz et al. [9] conduct a parallel "shadow execution" of a program with higher precision "shadow values" to detect inaccuracies. Fu et al. [20] automatically perform backward error analyses on numerical code. Fu and Su [22] propose a floating-point analysis via a reduction theory from any given analysis objective to a problem of minimizing a floating-point function.

When it comes to testing numerical libraries (as opposed to individual functions, either standing alone or within a larger program context), many of the above works and many others like [33, 41, 42] evaluate the effectiveness of their tools on portions of popular numerical libraries such as `GSL` and LAPACK, but none has applied a differential approach to compare such libraries.

Of the papers cited in this work, only Dutta et al. [18] focus their efforts on numerical libraries written in Python, namely the probabilistic programming libraries Edward and Stan. Though the work of Yi et al. [48] focuses on 20 special functions in `GSL`, they do have a small aside that applies their tool to 16 analogous functions discovered manually in `SciPy`. In addition, they also use analogous `mpmath` functions executed at high-precision as their ground truth.

Differential Testing. Differential testing [34] tests multiple implementations that share the same functionality with the same input. The goal is to identify bugs by observing the output differences. Its utility as a tool for finding bugs has been well-documented, being used to test compilers [40, 46], JVM implementations [13], program analyzers [29], probabilistic programming systems [18], interactive debuggers [31], code coverage tools [47], and certificate validation in SSL implementations [14] to name a few.

In its built-in tests, `SciPy` conducts a manually written differential test of a subset of special functions against the analogous functions in the `mpmath` library [26]. This manual effort highlights the value of differential testing in the context of numerical libraries. To the best of our knowledge, we are the first to conduct automated differential testing of numerical libraries for the purpose of discovering discrepancies among libraries.

Test Migration. Behrang and Orso [8] introduce a tool for automatically migrating tests across similar mobile applications, essentially translating a sequence of events into a form that can be consumed by the target app. Qin et al. [38] present TestMig, an approach to migrate GUI tests from iOS to Android. To the best of our knowledge, we are the first to use test migration in the context of numerical software.

API Mapping. Nguyen et al. [35, 36] use supervised learning to train a neural network on existing code migrations for API mapping. Bui [11] proposes to use Generative Adversarial Networks to "align" two different vector spaces of embeddings generated separately for a pair of APIs. Recent work [17] creates function embeddings to find function “synonyms” across Linux file systems and drivers. However, to the best of our knowledge, such techniques have not been applied to the APIs of numerical libraries. The former approach suffers from a lack of parallel corpora. The latter two utilize embeddings of functions based on their calling context which, for numerical functions, can be widely-varied or even non-existent if being used for one-off calculations.

In the realm of mapping math APIs, Santhiar et al. [39] similarly leverage the idea of mining function specifications from unit tests and using the outputs of functions to produce function mappings. However, their tool MathFinder requires the user to input a description of the desired functionality. This query is then tested against the unit tests of the target library to find matching outputs. MathFinder's motivation is different; mappings are targeted and queries are formed one at a time manually. By contrast, our technique is automatic, untargeted, and it does not require input specifications.

6 CONCLUSION

We proposed an approach for finding discrepancies between synonymous functions in different numerical libraries as a means of identifying incorrect behavior. Our approach automatically finds such synonymous functions, synthesizes testing drivers, and executes differential tests to determine meaningful discrepancies across numerical libraries. We implemented our approach in a tool named FPDiff, which automatically discovered 126 equivalence classes across the libraries `GSL`, `jmat`, `mpmath`, and `SciPy` with a 95.8% precision and 79.0% recall. The fact that discrepancies are inherently difficult to find enabled us to cluster function synonyms based on their outputs over *elementary inputs*. Using a selection of *adversarial inputs*, we discovered 655 unique discrepancies within equivalence classes of these function synonyms, 150 of which represent 125 unique bugs. We have reported all bugs to library developers; so far, 30 bugs have been fixed, 9 have been found to be previously known, and 25 more have been acknowledged by developers.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1750983.

REFERENCES

[1] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. 1990. LAPACK: A Portable Linear Algebra Library for High-performance Computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing (Supercomputing '90)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 2–11. http://dl.acm.org/citation.cfm?id=110382.110385

[3] Douglas N Arnold. 2000. The Explosion of the Ariane 5. (2000). http://www-users.math.umn.edu/~arnold/disasters/ariane.html

[4] Douglas N Arnold. 2000. The Patriot Missile Failure. (2000). http://www-users.math.umn.edu/~arnold/disasters/patriot.html

[5] Roberto Bagnara, Matthieu Carlier, Roberta Gori, and Arnaud Gotlieb. 2013. Symbolic Path-Oriented Test Data Generation for Floating-Point Programs. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 1–10. https://doi.org/10.1109/ICST.2013.17

[6] Tao Bao and Xiangyu Zhang. 2013. On-the-fly detection of instability problems in floating-point program execution. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 817–832. https://doi.org/10.1145/2509136.2509526

[7] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 549–560. https://doi.org/10.1145/2429069.2429133

[8] Farnaz Behrang and Alessandro Orso. 2018. Automated test migration for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 384–385. https://doi.org/10.1145/3183440.3195019

[9] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A dynamic program analysis to find floating-point accuracy problems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 453–462. https://doi.org/10.1145/2254064.2254118

[10] Phelim Boyle and Alex Potapchik. 2006. Application of high-precision computing for pricing arithmetic asian options. In *Symbolic and Algebraic Computation, International Symposium, ISSAC 2006, Genoa, Italy, July 9-12, 2006, Proceedings*, Barry M. Trager (Ed.). ACM, 39–46. https://doi.org/10.1145/1145768.1145782

[11] Nghi D. Q. Bui. 2019. Towards zero knowledge learning for cross language API mappings. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019.*, Gunter Mussbacher, Joanne M. Atlee, and Tevfik Bultan (Eds.). IEEE / ACM, 123–125. https://dl.acm.org/citation.cfm?id=3339722

[12] Tsong Yueh Chen, Fei-Ching Kuo, T. H. Tse, and Zhiquan Zhou. 2003. Metamorphic Testing and Beyond. In *11th International Workshop on Software Technology and Engineering Practice (STEP 2003), 19-21 September 2003, Amsterdam, The Netherlands*. IEEE Computer Society, 94–100. https://doi.org/10.1109/STEP.2003.18

[13] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1257–1268. https://doi.org/10.1109/ICSE.2019.00127

[14] Yuting Chen and Zhendong Su. 2015. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 793–804. https://doi.org/10.1145/2786805.2786835

[15] Wei-Fan Chiang, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. 2015. Practical Floating-Point Divergence Detection. In *Languages and Compilers for Parallel Computing - 28th International Workshop, LCPC 2015, Raleigh, NC, USA, September 9-11, 2015, Revised Selected Papers (Lecture Notes in Computer Science)*, Xipeng Shen, Frank Mueller, and James Tuck (Eds.), Vol. 9519. Springer, 271–286. https://doi.org/10.1007/978-3-319-29778-1_17

[16] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, José E. Moreira and James R. Larus (Eds.). ACM, 43–52. https://doi.org/10.1145/2555243.2555265

[17] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to error-handling specification mining, See [30], 423–433. https://doi.org/10.1145/3236024.3236059

[18] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing probabilistic programming systems, See [30], 574–586. https://doi.org/10.1145/3236024.3236057

[19] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 509–519. https://doi.org/10.1109/ASE.2017.8115662

[20] Zhoulai Fu, Zhaojun Bai, and Zhendong Su. 2015. Automated backward error analysis for numerical code. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 639–654. https://doi.org/10.1145/2814270.2814317

[21] Zhoulai Fu and Zhendong Su. 2017. Achieving high coverage for floating-point code via unconstrained programming. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 306–319. https://doi.org/10.1145/3062341.3062383

[22] Zhoulai Fu and Zhendong Su. 2019. Effective floating-point analysis via weak-distance minimization. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 439–452. https://doi.org/10.1145/3314221.3314632

[23] Amparo Gil, Javier Segura, and Nico M. Temme. 2007. *Numerical methods for special functions.* SIAM. https://doi.org/10.1137/1.9780898717822

[24] Brian Gough. 2009. *GNU Scientific Library Reference Manual - Third Edition* (3rd ed.). Network Theory Ltd.

[25] Hui Guo and Cindy Rubio-González. 2020. Efficient Generation of Error-Inducing Floating-Point Inputs via Symbolic Execution (To Appear). In *International Conference on Software Engineering (ICSE)*.

[26] Fredrik Johansson et al. 2013. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18)*. `http://mpmath.org/`.

[27] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001–. SciPy: Open source scientific tools for Python. (2001–). http://www.scipy.org/

[28] Upulee Kanewala and James M. Bieman. 2013. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*. IEEE Computer Society, 1–10. https://doi.org/10.1109/ISSRE.2013.6698899

[29] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 239–250. https://doi.org/10.1145/3293882.3330553

[30] Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). 2018. *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM. http://dl.acm.org/citation.cfm?id=3236024

[31] Daniel Lehmann and Michael Pradel. 2018. Feedback-directed differential testing of interactive debuggers, See [30], 610–620. https://doi.org/10.1145/3236024.3236037

[32] A. D. MacDonald and Sanborn C. Brown. 1949. High Frequency Gas Discharge Breakdown in Helium. *Phys. Rev.* 75 (Feb 1949), 411–418. Issue 3. https://doi.org/10.1103/PhysRev.75.411

[33] Osni Marques, Christof Vömel, James Demmel, and Beresford N. Parlett. 2008. Algorithm 880: A testing infrastructure for symmetric tridiagonal eigensolvers. *ACM Trans. Math. Softw.* 35, 1 (2008), 8:1–8:13. https://doi.org/10.1145/1377603.1377611

[34] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107. http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[35] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2016. Mapping API elements for code migration with vector representations. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 756–758. https://doi.org/10.1145/2889160.2892661

[36] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 438–449. https://doi.org/10.1109/ICSE.2017.47

[37] Michael Pradel and Koushik Sen. 2018. DeepBugs: a learning approach to name-based bug detection. *PACMPL* 2, OOPSLA (2018), 147:1–147:25. https://doi.org/10.1145/3276517

[38] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. TestMig: migrating GUI test cases from iOS to Android. In *Proceedings of the 28th ACM SIGSOFT International*

Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019., Dongmei Zhang and Anders Møller (Eds.). ACM, 284–295. https://doi.org/10.1145/3293882.3330575

[39] Anirudh Santhiar, Omesh Pandita, and Aditya Kanade. 2014. Mining Unit Tests for Discovery and Migration of Math APIs. *ACM Trans. Softw. Eng. Methodol.* 24, 1 (2014), 4:1–4:33. https://doi.org/10.1145/2629506

[40] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and analyzing compiler warning defects. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 203–213. https://doi.org/10.1145/2884781.2884879

[41] Enyi Tang, Earl T. Barr, Xuandong Li, and Zhendong Su. 2010. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, Paolo Tonella and Alessandro Orso (Eds.). ACM, 131–142. https://doi.org/10.1145/1831708.1831724

[42] Enyi Tang, Xiangyu Zhang, Norbert Th. Müller, Zhenyu Chen, and Xuandong Li. 2017. Software Numerical Instability Detection and Diagnosis by Combining Stochastic and Infinite-Precision Testing. *IEEE Trans. Software Eng.* 43, 10 (2017), 975–994. https://doi.org/10.1109/TSE.2016.2642956

[43] Lode Vandevenne. 2014. jmat: Complex special functions, numerical linear algebra and statistics in JavaScript. https://github.com/lvandeve/jmat. (2014).

[44] Ran Wang, Daming Zou, Xinrui He, Yingfei Xiong, Lu Zhang, and Gang Huang. 2016. Detecting and fixing precision-specific operations for measuring floating-point errors. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 619–630. https://doi.org/10.1145/2950290.2950355

[45] Eric W Weisstein. [n. d.]. Special Function. ([n. d.]). http://mathworld.wolfram.com/SpecialFunction.html

[46] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532

[47] Yibiao Yang, Yuming Zhou, Hao Sun, Zhendong Su, Zhiqiang Zuo, Lei Xu, and Baowen Xu. 2019. Hunting for bugs in code coverage tools via randomized differential testing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 488–498. https://doi.org/10.1109/ICSE.2019.00061

[48] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient automated repair of high floating-point errors in numerical libraries. *PACMPL* 3, POPL (2019), 56:1–56:29. https://doi.org/10.1145/3290369

[49] Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. 2015. A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 529–539. https://doi.org/10.1109/ICSE.2015.70