Floating-Point Usage on GitHub: A Large-Scale Study of Statically Typed Languages

ANDREA GILOT, Uppsala University, Sweden
TOBIAS WRIGSTAD, Uppsala University, Sweden
EVA DARULOVA, Uppsala University, Sweden

Reasoning about floating-point arithmetic is notoriously hard. While static and dynamic analysis techniques or program repair have made significant progress, more work is still needed to make them relevant to real-world code. On the critical path to that goal is understanding what real-world floating-point code looks like.

To close that knowledge gap, this paper presents the first large-scale empirical study of floating-point arithmetic usage across public GitHub repositories. We focus on statically typed languages to allow our study to scale to millions of repositories. We follow state-of the art mining practices including random sampling and filtering based on only intrinsic properties to avoid bias, and identify floating-point usage by searching for keywords in the source code, and programming language constructs (*e.g.,* loops) by parsing the code. Our evaluation supports the claim often made in papers that floating-point arithmetic is widely used. Comparing statistics such as size and usage of certain constructs and functions, we find that benchmarks used in literature to evaluate automated reasoning techniques for floating-point arithmetic are in certain aspects representative of 'real-world' code, but not in all.

We publish a dataset of 10 million real-world floating-point functions extracted from our study. We demonstrate in a case study how it may be used to identify new floating-point benchmarks and help future techniques for floating-point arithmetic to be designed and evaluated to match actual users' expectations.

CCS Concepts: • **Software and its engineering** → **Software libraries and repositories**; **Automated static analysis**; • **Information systems** → *Data mining*.

Additional Key Words and Phrases: floating-point arithmetic, large-scale code analysis, repository mining

ACM Reference Format:
Andrea Gilot, Tobias Wrigstad, and Eva Darulova. 2026. Floating-Point Usage on GitHub: A Large-Scale Study of Statically Typed Languages. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 95 (April 2026), 28 pages. https://doi.org/10.1145/3798203

1 Introduction

Numerical software is widely used across different domains such as embedded systems, scientific computing, and machine learning. Such software frequently makes use of floating-point arithmetic to efficiently approximate computations with real numbers. Some approximation of the infinitely precise reals is fundamentally and practically necessary, and introduces rounding errors at most arithmetic operations as well as potentially the special values infinity and Not-a-Number (NaN). Overall this makes reasoning about the correctness of such software unintuitive, error prone and time consuming.

This could be the beginning of a paper on automated reasoning for floating-point programs, *e.g.,* with a static [19, 53] or a dynamic approach [31, 63, 65]. Such an approach would typically be

Authors' Contact Information: Andrea Gilot, Uppsala University, Uppsala, Sweden, andrea.gilot@it.uu.se; Tobias Wrigstad, Uppsala University, Uppsala, Sweden, tobias.wrigstad@it.uu.se; Eva Darulova, Uppsala University, Uppsala, Sweden, eva.darulova@it.uu.se.

ACM 2475-1421/2026/4-ART95
https://doi.org/10.1145/3798203

evaluated either using a small benchmark suite [18], or on a relatively small set of benchmarks hand-picked specifically for that paper.

While hand-picked benchmarks can be helpful to demonstrate particular features and issues, it is unclear how they generalise to "real-world" code. To the best of our knowledge, no large-scale code study has investigated how representative the currently used floating-point benchmarks are of "real-world" usage, *e.g.,* in terms of size, operations performed, or frequency of library function usage. Neither do we know how prevalent the use of floating-point arithmetic is in modern software.

While real-world floating-point usage is anecdotal, the difficulty in developing automated reasoning techniques for floating-point arithmetic is well documented. For example, static analysis techniques bounding worst-case rounding errors [19, 53, 58] are limited to small programs (mostly individual functions) and have no or limited support for common program constructs such as conditionals and loops. This is also reflected in the standard benchmark set [18] used in this domain. Dynamic approaches aiming to identify specific inputs that induce large rounding errors [31, 65], or for repairing such errors [47, 63] are evaluated on different, hand-picked sets of benchmarks, most of which are typically small (*e.g.,* individual functions). In summary, the current benchmarks used align with the current capabilities of the tools, but it is unclear to what extent they align with actual users' expectations in terms of the code they want the tools to reason about. While community benchmarks are a great way of focusing research, unless the benchmarks are "relevant", they may end up directing research in the wrong direction.

This paper presents the first *large-scale* study of real-world floating-point code. The aim is to inform current and future research efforts in this domain, for example by aiding (more) representative selections of benchmarks for evaluation, and to guide the development of future benchmarks and reasoning techniques in a practically relevant direction. With that in mind, we address the following research questions:

RQ1. How prevalent is floating-point arithmetic in open-source statically typed code on GitHub?

RQ2. What are the characteristics of floating-point code in terms of programming language constructs and size?

RQ3. How representative are benchmarks used in the literature to evaluate floating-point analyses of real-world floating-point code?

Characterising "real-world floating-point code" is non-trivial for several reasons. First, limiting ourselves to publicly available code, real-world code bases use a multitude of programming languages and the volume is too large to be processed and inspected in full. The latter can be addressed by sampling, but to avoid measurement bias, the code must be checked for code clones [42], and filtered to exclude projects that do not contain any code at all and "toy" projects (*e.g.,* projects that were never updated after the initial commit). While a convenient and continuously updated dataset of GitHub projects [16] aimed for code studies as ours exists, it includes only projects with at least ten stars (to reduce the amount of data). Such filtering has been shown to skew the results significantly [44], making it unsuitable to generalise from.

Second, once we have a set of "interesting" projects, there is no existing tool that can efficiently *identify* and *analyse* floating-point code. To obtain representative results, such an analysis must support a multitude of programming languages and be automated to be able to handle sufficiently large volumes of code. Our insight is that in *statically typed* languages the type annotations on *e.g.,* function parameters allow us to identify floating-point code with an inexpensive and largely programming language-agnostic approach by "grepping" for suitable keywords in source code. To identify floating-point code, *type* annotations are critical; it is not enough to look for the presence of arithmetic operators (*e.g.,* +) as these are often overloaded to work for integers, strings or other types.

This is unlike other code properties such as imports of specific libraries that can be unambiguously identified even in dynamically typed source code.[1] In this paper, we therefore limit ourselves to studying floating-point usage in statically typed languages.

Contributions. This paper makes the following contributions:

Large-Scale Floating-Point Code Mining. We design and implement a large-scale mining methodology (Section 3) for detecting and analysing floating-point usage in statically typed languages at the project, file and function level. Our approach is fully automated and combines random sampling, project filtering based on intrinsic properties, and fuzzy duplicate elimination to produce a reliable dataset.

Our methodology identifies floating-point usage in statically typed programming languages using *keywords* in the source code directly. This allows us to consider *all* statically typed languages that are recognised by GitHub in a scalable and fully automated way, as long as these languages have floating-point types and a language reference. For the most commonly used languages (covering more than 92 % of the sampled code), we parse the code to identify programming language constructs (*e.g.,* loops) using a language-agnostic parsing library.

Large parts of the methodology and implementation are not specific to floating-point analysis (nor to statically typed code, although we rely on types for identifying relevant code) and we expect them to be reusable in other large-scale empirical code studies. The code is available as open-source[2].

Quantitative Analysis. We analyse 447 209 (randomly sampled) GitHub projects (Section 4) and present the results in Section 5. Our analysis shows that, with 95 % confidence, over 62 % of the projects contain floating-point code, supporting the often made claim that floating-point numbers are widely used. While this confirms folklore beliefs, we are the first to draw this conclusion based on a large-scale code study of open-source code.

Most functions in our final dataset tend to be small, and function calls and conditional statements appear more often than loops and (explicitly mentioned) special values. The FPBench [18] benchmark suite used in literature to evaluate floating-point techniques shows different characteristics, *e.g.,* involving fewer conditionals and more transcendental library function calls. Functions from the GNU Scientific Library, also sometimes used in evaluations, appear rarely in our dataset.

To the best of our knowledge, our study is the first to provide a clear picture of how floating-point numbers are actually used in real-world code, and indicates that currently used floating-point benchmarks are largely not representative of such code.

Floating-Point Functions Dataset and Challenge Benchmarks. For the benefit of other researchers, we release a dataset of 10 million real-world floating-point functions extracted from our study[3]. This dataset focuses on functions, since most static and dynamic floating-point reasoning tools today work on a per-function basis, so this dataset is most (immediately) relevant for their benchmarking. We intend this dataset to serve as a foundation for constructing new realistic benchmarks and evaluating floating-point reasoning tools in a practically relevant way, and invite others to contribute in this undertaking.

We show how our corpus can be used to generate realistic, ready-to-run benchmarks that reflect real-world floating-point practice via a case study (Section 6) that extracts 59 self-contained C floating-point benchmarks from functions in our dataset.

[1] Our study shows that mathematical library functions (*e.g.,* `sin`, `exp`) cannot be used to identify floating-point code either, as much floating-point code does not use them.

[2] Available at: https://github.com/fxpl/scyros.

[3] Available at https://doi.org/10.5281/zenodo.17055622.

2 Related Work

Floating-Point Reasoning Benchmarks. Most works that focus on numerical programs are evaluated using different sets of hand-picked programs. Hand-picking programs can be good for stressing certain types of problems, but can just as easily—intentionally or not—suppress other problems and challenges. It is also unclear how to generalise from hand-picked problems to code "in the real world". (Indeed, this was part of what prompted this research in the first place.)

We are aware of only a single benchmark suite created specifically for floating-point analysis: FPBench [18]. FPBench is typically used to evaluate static analysis tools [58]. The FPBench benchmarks were collected from individual papers [19, 53] where they were hand-picked and are typically isolated arithmetic expressions. They are therefore not likely representative of challenges found in real-world programs. Techniques that target programs outside of this set choose their own set of benchmarks, *e.g.,* programs that are larger [20] or that use specific data structures [34].

Some techniques are evaluated on a small number of hand-picked functions from numerical libraries, *e.g.,* from the GNU Scientific Library (GSL) [31, 46, 63, 65], occasionally extended by some additional functions. For example, Di Franco et al. [23] study bug characteristics in numerical libraries (NumPy, LAPACK, GSL, etc.), and Liew et al. [41] have two teams independently hand-pick benchmarks for evaluating symbolic execution. In the HPC community, there are examples of picking a specific proxy application, such as Lulesh, as a representative of "HPC code" [61]. The tool NSan [15] is evaluated on the commercial SPECfp2006 benchmark set that contains scientific computing applications and that was designed for performance benchmarking. In what way these selected benchmarks are representative of general real-world code, *e.g.,* beyond just library code or a specific application domain, is unclear and not discussed in these papers.

Real-World Usage through Code Studies. We are not aware of any large-scale code study that investigates characteristics of real-world floating-point code. Previous works have investigated how different features are used in practice in different programming languages, for example in the context of static analysis for the R programming language [52], dynamic features in Smalltalk [12], the use of `eval` and dynamic behaviour in JavaScript [50, 51], dynamic features or code complexity in Python [6, 30], inheritance, method chaining, and streams in Java [25, 39, 40, 57], or gradual types in TypeScript and Python [59]. While this paper also studies features of code, these are sufficiently different that we cannot straightforwardly re-use existing infrastructure. We furthermore consider floating-point usage across many different programming languages.

While some features in dynamically typed programming languages such as Python can be detected statically [48], others—specifically types—require programs to be run on suitable inputs [6]. For example, to determine whether an addition (+) operates over floating-point values requires detection of the operand types at runtime. This cannot be straightforwardly automated; to run code it is necessary to obtain—for each project individually—all its required dependencies, and understand how to run the program in a representative way (including representative and meaningful input data and GUI interactions). Such information is not available in a consistent format that can be automatically processed at a large scale. We thus focus on statically typed programming languages only in this study.

GitHub Mining. Our methodology draws inspiration from state-of-the-art practices in GitHub mining. Prior studies show that the majority of GitHub repositories are inactive, personal, or not software projects at all, and that most of the files are duplicates of others [35, 36, 42]. Other work demonstrates that filtering repositories using extrinsic characteristics such as the number of stars introduces systematic biases in software repository studies [11, 44]. In a reproduction study, Berger et al. reveal the risks of "careless" keyword searches (*e.g.,* tagging commits as bug fixes if they contain the word `"fix"` without excluding `"infix"`, `"suffixes"`, or `"does not fix"` etc.) and emphasise the necessity of manual validation to ensure methodological soundness [9].

Several frameworks exist to simplify code studies on GitHub, but none fully satisfy our requirements. Queryable datasets mined from GitHub exist, yet they are either outdated [24], affected by inconsistencies [29], or rely on GitHub stars [16, 17] that have been shown to bias results [44]. World of Code [43] continuously aggregates repositories from multiple sources and could be used to implement parts of our methodology, but it does not expose part of the metadata we require and does not allow downloading of full projects. Other frameworks target different aspects of repository analysis. CodeDJ [45] ensures reproducible queries over projects by downloading full project histories, which exceeds our needs and would lead to an excessive amount of data being downloaded; the analysis would also still need to be implemented "from scratch". GitCProc [13] enables keyword searches in commit histories but supports only a limited set of languages and does not exclude comments or string literals, which compromise validity. CodeQL [1] offers semantic analysis across multiple languages but requires full-project access, raising scalability concerns.

3 Methodology

Our goal is to study the prevalence and type of code operating on floating-point numbers in statically typed "real-world" code. To that end, we design a methodology that mainly follows the approach suggested by Maj et al. [44] and that is designed in accordance with the ACM SIGSOFT Empirical Standards for repository mining [49]. We first give an overview here, before discussing details in the next subsections.

3.1 Overview

We first define our *target population*: functions from non-trivial projects hosted on GitHub containing floating-point computations within code written in statically typed languages. GitHub alone contains an enormous amount of public repositories, and we assume here that they form a representative sample of real world software[4]. We restrict our study to statically typed code so that we can automate the analysis and thus to scale to significantly more projects.

Our final dataset consists of functions that use floating-point arithmetic. We focus on individual functions as most floating-point reasoning tools work on a per-function basis, though the corresponding files and projects are easily identified. Our quantitative analysis considers per-project, per-file and per-function properties, and so code not appearing in functions will be considered in the first two.

We cannot query GitHub for non-trivial projects containing such functions directly in part because each GitHub API call provides only limited information. We thus design a multi-step selection process or pipeline that progressively reduces the amount of data, increasing the level of granularity from the repository to the file and ultimately to the function level. An overview of the data selection pipeline is shown in Figure 1.

We begin by uniformly sampling public repositories on GitHub using GitHub's Rest API (step 1), and by excluding forks and 'trivial' projects (defined in Section 3.2, step 2 and 3). For each remaining repository, we query the list of used programming languages and retrieve the hash of the last commit on the main branch (step 4). We discard projects that do not have any files written in statically typed languages (step 5). *At the end of this step, we have metadata about the repositories, after preliminary filtering.*

[4]Although our methodology and implementation use GitHub as the primary data source, they are not specific to it. Both can be extended or adapted to other hosting platforms.

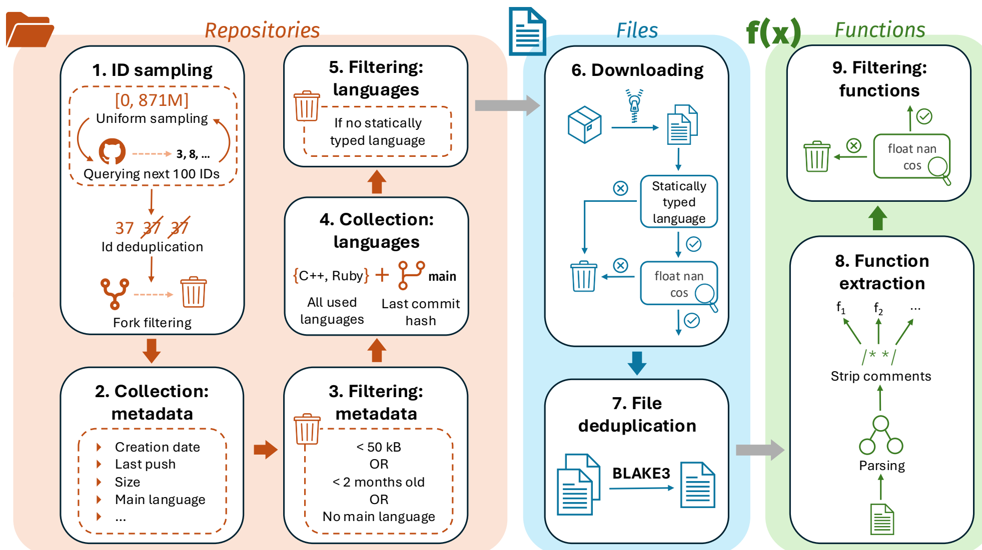

Fig. 1. Multi-step methodology for identifying and analysing floating-point code

Next, we download the content of the selected repositories and inspect individual files within these that are written in statically typed programming languages (step 6). For these we identify floating-point usage using keywords in the source code directly, *e.g.*, focusing on type annotations (*e.g.*, `double` and `real`) or mathematical functions (*e.g.*, `sin` and `exp`). We discard data that we do not consider: files that are not written in statically typed languages, files that do not contain any keyword related to numerical computations, and files that are token-level duplicates of other files (in any considered repository, step 7). We then parse the remaining files, remove comments, and extract individual functions (step 8). We retain only functions that include keywords related to floating-point computations (step 9). In the final step we quantify usage of programming language constructs such as loops or conditional statements fully automatically based on parse trees.

The pipeline is designed to be fully automated, reproducible, modular and extensible. The sampling and filtering steps earlier in the pipeline are not specific to floating-point arithmetic and can be re-used for other code studies. We envision the floating-point specific dataset collected at the end of the pipeline to be used, *e.g.*, to sample or validate benchmarks with specific features that are used for evaluating new floating-point automated reasoning techniques.

We now expand on the details of each step of the pipeline and choices made during the design of the methodology.

3.2 Project Selection

We sample ids of public repositories on GitHub (step 1) uniformly using GitHub's REST API. Given an id, the API returns the next 100 ids of public repositories. To sample uniformly, we get the largest project id at the beginning of data collection[5], and generate a random integer between 0 and this maximal id to get the next following 100 ids. The random number generator we use for sampling has a fixed seed to ensure reproducibility. In addition to each project id, the API call returns the name of each repository and whether it is a fork. Since we are sampling with replacement, we end

[5]At the time of writing, GitHub assigns repository ids in chronological order. To obtain the current maximal id, we create a new repository and query its id; the maximum id for our current dataset was obtained on 24 January 2025.

Ada, AngelScript, Arduino, ATS, Ballerina, Beef, Boo, C, C#, C++, Chapel, Clean, Crystal, Curry, CWeb, EC, Eiffel, Elm, F#, F, Fantom, Fortran, FreeBasic, Frege, Futhark, Gleam, Go, Haxe, Haskell, Java, Kotlin, Modula-2, Modula-3, Nemerle, Nim, Standard ML, Oberon, OCaml, Opa, Pascal, Reason, Rust, Purescript, Scala, Swift, Typescript, Vala, Uno, Volt, Xtend, and Zig.*

Fig. 2. List of the 51 programming languages we include in this study.

up with duplicates that we remove after the sampling. We also remove forks from the dataset as they bias the results of the study.

We collect metadata (step 2) for the sampled repositories using GitHub's REST API. The metadata includes the size of the repository, its creation date, last push date, number of stars, number of issues, and its primary programming language. As noted by Kalliamvakou et al. [36], one of the challenges in mining GitHub is that a large portion of repositories are not software projects. In addition, most of them are inactive and contain very few commits. Our target population consists of *nontrivial* or "non-toy" software projects, but defining what is nontrivial is difficult and subject to the authors' interpretation. As Maj et al. [44] suggest, it is easier to discard projects we know are *not* part of the population than to define what is in it.

We exclude a project from our dataset if any of the following hold (step 3):

- it is smaller than 50 kB,
- GitHub does not assign a primary language to it (these are typically templates, or projects containing only documentation or binary files),
- the creation date and last activity date are less than two months apart (so the "age" of the project is less than two months).

The rationale for using age as an exclusion criterion is to avoid including repositories in the population that were *e.g.,* used for student assignments. An alternative or complement to the age is the number of commits, though in practice querying the number of commits using the API is expensive.

We do not filter based on social engagement, *i.e.,* stars or issue counts, as it is unclear whether analysing only these repositories would generalise [11, 44]. While the number of contributors could be used as a filter, we choose not to as single-maintainer projects could still reasonably be part of the population.

3.3 Language Filtering

Our target population consists of code written in statically typed programming languages. Since the metadata we collect in the previous step includes only the primary language of each repository, we perform an additional query to retrieve the list of all the languages used in each project (step 4). As it is the last step before downloading repositories, we also query the hash of the latest commit on the main branch. This allows us to ensure the reproducibility of the results: if the repository remains reachable and its commit history has not been rewritten, it can be re-downloaded in the exact same state.

For each programming language recognised by GitHub, we determine whether it is statically typed (based on its documentation) and, if so, include it in our list of target languages. We exclude languages that do not have primitive floating-point types (*e.g.,* Coq, Agda, or Dafny). We also exclude languages where we could not find the language reference (*e.g.,* Mirah or ActionScript). This yields the list of 51 statically typed programming languages in Figure 2. We discard repositories that do not contain code written in at least one of them (step 5).

3.4 Downloading and Keyword Filtering

We download (step 6) the content of the filtered repositories using the hash of the latest commit on the main branch we retrieved in the previous step. We exclude files not written in a statically typed language. We determine programming language using file extensions, as repositories do not contain metadata for mapping language to files. We then compute the number of lines of code and the number of words.

We aim to retain files with functions operating on floating-point values. The most precise approach to identify these would be to infer the type of each individual operation, *i.e.,* run a type checker or type inference. This approach, however, would require special treatment and significant effort for each individual programming language. We instead choose an approximate, but more scalable, approach, and aim to identify functions with floating-point operations based on keywords used in the source code via regular expressions. Because the programming languages we are dealing with are all statically typed, we know that function signatures often contain the types of the parameters and return values. Hence, we search for floating-point types in the files using a regular expression. A problem encountered by this approach is the presence of type aliases, structures or classes that represent or contain floating-point values and that might be defined in other files. To mitigate this issue, we also look at other keywords related to floating-point computations, such as transcendental functions, rounding functions, and special values like NaN or infinity. Looking at the presence of transcendental functions is also interesting per se, as they are a feature that several floating-point verification tools aim to support. Finally, we examine the use of keywords related to arbitrary-precision floating-point computations to assess how frequently such features are used in practice.

For each of the 51 languages, we manually inspect the official language reference to identify floating-point types and special values, and we examine the corresponding standard math library to identify numerical functions. We group keywords into four categories:

- Primitive floating point types: keywords used to declare standard floating point types, *e.g.,* `float`, `double`, `real`, `float32`, `float64`.
- Transcendental functions: keywords used to call transcendental functions, *e.g.,* `sin`, `cos`, `tan`, `exp`, `log10`.
- Other keywords related to floating point computations and special values, *e.g.,* `fma`, `posinf`, `nan`, `round`, `ulp`.
- Arbitrary precision floating-point functions and types, *e.g.,* `mpfr_t`, `bigfloat`, `apfloat`.

The JSON files containing the keyword definitions for each programming language are available along with our function dataset.

The search is case insensitive using boundaries such as spaces, punctuation or brackets to surround the words. While choosing keywords, we tried to be as exhaustive as possible while avoiding ambiguous keywords. For example, keywords like `abs` or `pow` are valid floating-point functions but they are often used in integer computations as well, and could lead to false positives. While some keywords like `cos`, `sin` or `float` are common to most languages, others can be specific to a given programming language and ambiguous in others. This is the case for the keyword `number` in Typescript, which may denote floating-points or integers, or `real` in Pascal or Fortran. For each file, we count the number of keyword occurrences per category, and discard the file if it does not contain any keyword from neither of the four categories.

3.5 De-Duplication

Removing duplicates is a crucial step to avoid measurement bias, as prior work shows that GitHub code contains considerable amounts of code duplication [42]. We eliminate files that are identical up to token reordering across repositories (step 7). For this step and the following ones, we abandon the notion of repositories and treat files as standalone entities within a global pool. We represent each file as a bag-of-words, ignoring whitespace, delimiters, special characters, and token order. This representation enables the detection of token-level duplicates that differ only by whitespace changes or reordered declarations. We compute the BLAKE3 hash of each bag-of-words representation and retain a single instance for each unique hash.

3.6 Parsing and Function Extraction

Once only deduplicated files containing numerical keywords are left, we parse them to extract individual functions (step 8). We remove comments because they are not directly relevant to our analysis, may be outdated, and can bias results. In particular, terms such as double may appear in natural language text with meanings unrelated to floating-point types, thereby increasing the risk of false positives. Comments are clearly valuable in their own right and deserve special care, but we leave their analysis for future work.

We traverse the parse tree to extract individual functions. For each function, we compute its metadata as the number of:

- lines of code;
- words[6];
- parameters (*i.e.,* the function's arity);
- numerical types parameters;
- conditional statements in the body of the function, and their maximum nesting depth;
- loops in the body of the function, and their maximum nesting depth;
- function calls in the body of the function, and their maximum nesting depth (*e.g.,* `sin(cos(x))` has depth 2);
- occurrences of numerical keywords in the function.

We also record the function's location in the source file and whether a parsing error occurred, including its position when applicable. Similar to comments, we exclude string literals from the keyword search, since keywords occurring in free text are more likely to produce false positives. If a function contains at least one numerical keyword, we extract its code and store it in a separate file (step 9).

We parse the files using tree-sitter [2], a language agnostic parsing library that already comes with grammars for the most used programming languages. Our implementation currently supports C, C++, Java, TypeScript, C# and Go, which together account for more than 92 % of the functions collected. Supporting additional languages requires only defining the parse nodes of interest, which makes our implementation readily reusable and extendable in future code studies. While other statistics, such as the number of floating-point arithmetic operations, would have been relevant for the study, these rely on type information, which is beyond tree-sitter's capacity[7]; *i.e.,* they would require programming language-specific analyses.

4 Experimental Setup

Implementation. Our approach is implemented in Rust and designed as a general-purpose framework, called Scyros, for large-scale code studies, with a focus on reproducibility and reusability. Data collection can be interrupted and resumed at any time without compromising reproducibility.

[6] A word is a sequence of letters, numbers, or underscores delimited by boundaries (*e.g.,* brackets, punctuation, whitespaces, beginning and end of file).

[7] Tree-sitter preserves type information that is explicitly declared in source code, but does not do any type-checking, propagation or inference.

Inputs are randomly shuffled before any pipeline step to enable subsampling. All random seeds are fixed and can be user-defined. During language collection, the hash of the latest commit on the main branch is stored, ensuring that repository contents remain consistent across re-runs, provided it is still reachable and history was not rewritten.

Scyros can be straightforwardly adapted in a number of ways for different analyses. For example, users can choose to analyse local repositories instead of downloading data from GitHub. They can also specify which file extensions or programming language to retain in the dataset, and adapt the set of keywords for each programming language depending on the target code features of interest. Multiple configuration files can be supplied simultaneously, enabling parallel analyses with different settings. When downloading data, users have the option to either delete files that do not pass filtering criteria or keep them for future use. Each pipeline step produces a CSV file that serves as the input for the next step. All steps can be executed independently as separate subcommands via the command-line interface. Individual steps can be skipped or performed manually by editing their corresponding CSV files without affecting other stages of the pipeline. Furthermore, additional steps can be introduced without disrupting existing functionality. The code is fully documented, tested, and released as open-source software under the Apache 2.0 license. We conduct the data analysis in Python.

Data Collection. We run our pipeline on an Ubuntu 24.04 machine with 128 GB RAM, 12 CPU cores at 4.95 GHz, using Rust 1.85.0 and Python 3.13.9. We ran the different pipeline steps in an interleaved fashion. Since the exact running time of each pipeline step depends directly on the number of sampled projects, we report here approximate durations that are aimed to mostly illustrate the relative differences between the pipeline steps. Steps that rely on the GitHub API are rate-limited to 5000 requests per hour per token, hence the durations vary depending on the number of available tokens. For our experiments, we used 8 tokens.

Sampling IDs took approximately a week, metadata collection took 2 months, language collection took a few days, downloading the files took 3 days, removing duplicates took 10 minutes and parsing took 60 minutes. We collected the data between October 2024 and March 2025.

Project Metadata Filtering. We first provide statistics about steps 1–5 of our pipeline (see Figure 1), which are largely not specific to floating-point arithmetic, before discussing the floating-point specific results in Section 5.

After deduplication, random sampling yielded 125 234 729 unique GitHub repository IDs, of which 67 % (83 615 678) were non-fork repositories (step 1). Because GitHub returns 100 repository IDs per request, but metadata can only be retrieved one repository at a time, we subsampled the IDs to make metadata collection tractable. Metadata collection (step 2) for sampled project IDs yielded 11 924 148 project entries.

At the time of metadata collection, 4.6 % of the projects were no longer reachable, having been deleted or made private. Metadata for the remaining reachable projects is summarised in the first row of Table 1. For each statistic, the table reports the minimum, median, and maximum values, as well as the first (Q1) and third (Q3) quartiles.

In summary, the dataset is skewed towards small, recent and inactive projects. More than 75 % of the projects are younger than two weeks, and 85 % younger than two months. Empty repositories account for 19 % of the dataset, while 49 % are smaller than 50 kB. Additionally, 12 % of the projects are non-empty but have no primary language assigned by GitHub, confirming the need for filtering. A small fraction (0.61 %) of the projects have a negative age, a consequence of GitHub relying on the local system time of the machine pushing the commit, which may be incorrectly configured.

After removing projects smaller than 50 kB, that were less than 2 months old, or contained no source code (step 3), we were left with 1 238 741 projects (11 % of the initially collected metadata).

Table 1. Distribution of project metadata before and after metadata filtering, and after keyword filtering. Q1 and Q3 denote the first and third quartiles. The gray row is discussed in Section 5.

	Metadata	Age	Creation (YY/MM)	Size (in kB)	# forks > 0	# forks among > 0	# issues > 0	# issues among > 0
All	Min	−19459 d	70/01	0		1		1
	Q1	1 min	20/02	2		1		1
	Median	36 min	22/02	56	6 %	1	6 %	2
	Q3	11 d	23/08	928		3		7
	Max	6104 d	25/01	109 M		144 k		70 k
Interesting	Min	60 d	08/02	50		1		1
	Q1	125 d	19/05	322		1		1
	Median	316 d	20/10	1836	20 %	2	31 %	5
	Q3	925 d	22/11	9759		5		14
	Max	6104 d	24/10	105 M		144 k		70 k
With FP keywords	Min	60 d	08/03	50		1		1
	Q1	119 d	19/01	372		1		1
	Median	290 d	20/12	2102	24 %	2	30 %	4
	Q3	844 d	22/12	14422		6		13
	Max	6090 d	24/10	42 M		27 k		15 k

We refer to these as the "interesting" projects in the second row of Table 1 which shows their metadata distribution. While the number of projects created on GitHub has grown exponentially, with the majority in the last five years, this trend is less pronounced in the filtered dataset. The median creation year of retained projects is two years earlier than that of the unfiltered dataset.

The dataset is dominated by projects written primarily in non-statically typed languages. According to GitHub's language classification, JavaScript accounts for the largest share at 22 %, followed by HTML (12 %) + CSS (4 %) and Python (10 %) + Jupyter Notebook (5 %). These proportions are similar before and after filtering, except for HTML, which drops from 17 % to 12 %. Among statically typed languages, the most common are Java (9 %), TypeScript (7 %) and C++ (4 %). After collecting the list of all languages used in each repository (step 4), and removing projects that do not include any statically typed language (step 5), we are left with 465 050 repositories, corresponding to 38 % of the filtered dataset. This dataset forms the basis for answering our research questions.

5 Quantitative Analysis of Floating-Point Usage

We address the following research questions to evaluate the results of our study and outline how we answer each of them:

RQ1. How prevalent is floating-point arithmetic in open-source statically typed code on GitHub? The aim is to confirm (or disprove) the conventional hypothesis, thus (in)validating the motivation for much past and future research.

We measure prevalence at both the project and file levels. First, we identify how many repositories and files contain floating-point code. At the file level, we further investigate whether floating-point operations are distributed throughout the file or concentrated in a few functions. We provide the complete answer to RQ1 on page 15.

Table 2. Proportion of projects, resp. files that contain certain types of keywords

Keyword category	Projects	Files		
		Q1	Median	Q3
Primitive floating-point types	64 %	7.1 %	17 %	35 %
Transcendental functions	20 %	1.3 %	3.5 %	9.5 %
Miscellaneous keywords	21 %	1.4 %	3.1 %	7.1 %
Arbitrary-precision keywords	1.1 %	0.13 %	1.4 %	3.6 %
Any	**66 %**	**7.5 %**	**18 %**	**36 %**

RQ2. What are the characteristics of floating-point code in terms of programming language constructs and size? The aim is to characterise "real-world" floating-point code to inform future floating-point tool development and benchmarking towards practical relevance.

We analyse the structure of floating-point functions in our dataset to identify commonly used programming language constructs, such as loops, branches, functions calls, and their combinations.

RQ3. How representative are benchmarks used in the literature to evaluate floating-point analyses of real-world floating-point code? We compare functions from FPBench [18] with those in our dataset based on their structural characteristics. We also examine how frequently functions from the GNU Scientific library that are used in the literature appear in real-world code.

In the following, unless otherwise specified, all reported quantities are rounded to two significant digits to enhance readability.

5.1 Project-Based Analysis

At the time of downloading (step 6 in Figure 1), 3.8 % of the projects were unreachable, leaving 447 209 accessible projects. From these, we now consider only those projects that contain floating-point keywords. Metadata for these projects is summarised in the third (grey) row of Table 1. While not the same, the metadata is quite similar to the one of the “interesting” projects.

Table 2 shows the breakdown of the different floating-point keyword categories by projects and files. Floating-point keywords appears in 66 % of the projects. Most often these are floating-point types, which appear in 64 % of the projects. Transcendental functions and keywords such as `NaN` appear less frequently explicitly in the code, while arbitrary-precision keywords appear very rarely (1.1 % of the projects).

For each project containing at least one floating-point keyword from a category, we compute the proportion of statically typed files containing at least one keyword from that category. The median and quartiles of these proportions are shown in the second column of the table. What we observe is that when floating-point related keywords are present in a project, they tend to be concentrated in a small number of files. In half of the projects that contain floating-point types, fewer than 17 % of the files include such keywords. This proportion is even lower for transcendental functions as well as miscellaneous and arbitrary-precision keywords.

We note that we compute these statistics before deduplication. This means that these statistics will likely contain some library code for projects but not for all, depending on which code has been committed to the repository.

5.2 File-Based Analysis

Presence of Keywords. After downloading the repositories and retaining only the files written in statically typed languages with floating-point keywords, we are left with 13 167 648 files. To estimate the false positive rate, we draw a random sample from this dataset and manually inspect each file to determine whether it contains actual floating-point code. Using Cochran's formula [14] for sample size estimation in large populations, we select 385 files, ensuring statistical representativeness at a 95 % confidence level with a 5 % margin of error. Of these 385 files, 314 contain valid floating-point code, while 71 are false positives. Among the false positives, 50 arise from keywords appearing in comments, 9 from keywords in string literals, 7 from keywords used in variable names, and 5 from other miscellaneous causes. In most cases, the keyword double appears in comments in C and C++ files. This indicates the importance of stripping comments and ignoring string literals before performing keyword-based detection at the function level in later steps of the methodology.

Given the results of the manual inspection and using Wilson score intervals [7], we estimate that the proportion of true positives in our dataset lies between 77 % and 86 %, with 95 % confidence. Assuming that the presence of false positives in files is independent across files, the probability that project i contains at least one file with actual floating-point code is

$$P_i = 1 - (1 - p)^{k_i}$$

where p is the estimated true positive rate, and k_i is the number of files with floating-point keywords in project i. Note that the independence assumption does not fully reflect reality, as files within the same project are not independent. Taking the expectation of P_i over all projects, we estimate that between 62 % and 64 % of the projects in our dataset contain floating-point code, with 95 % confidence.

That said, some files may have been missed due to the use of type-aliases or user-defined wrappers, so the numbers may be even higher. To assess the extent of this issue, we randomly sample 385 projects and manually inspect whether they redefine floating-point types. Among these, 22 projects do redefine floating-point types. Example of such aliases include declarations like `type money = number` in TypeScript, or conditional typedefs that map `float` and `double` to a common type `Float` to select precision at compile time with macros in C++. In all observed cases, these aliases are used alongside standard floating-point types rather than replacing them entirely, making the precise impact of this phenomenon difficult to quantify. An additional 10 projects define wrappers around floating-point values, typically simple structural wrappers in abstract syntax tree nodes. Over the full dataset, we therefore estimate with 95 % confidence that between 5.9 % and 11.5 % of projects are affected by floating-point type redefinitions or wrappers.

De-Duplication. A substantial fraction of the dataset is redundant, with 63 % of files identified as token-level duplicates and 61 % as exact duplicates. TypeScript exhibits the highest duplication rate, with declaration files duplicated on average 17 times and source files twice. C follows, with 3.4 copies per source file and 2.8 per header file. The most duplicated file originates from Angular, a TypeScript web application framework, and appears 27 136 times (0.70 % of TypeScript files in the dataset). Of the 100 most duplicated files, 98 are written in TypeScript. Collectively, these files account for 9.8 % of 3.9 million TypeScript files in the dataset. The most duplicated C files are mainly hardware drivers implementations.

Most Common Programming Languages. For parsing, we focus on the most commonly used languages in the dataset. However, using a single metric, such at the number of lines of code or files, to compare different programming languages is misleading. Some languages are inherently more verbose, while others require code to be split across multiple files. In addition, the rate of false positives varies by language. Therefore, we focus on the orders of magnitude, and avoid drawing conclusions based on exact numbers or ordering between languages with similar proportions.

To better understand each language's contribution and mitigate potential biases, we separate C and C ++ header files from C source files. We distinguish between C headers (`.h` and `.inc`), which

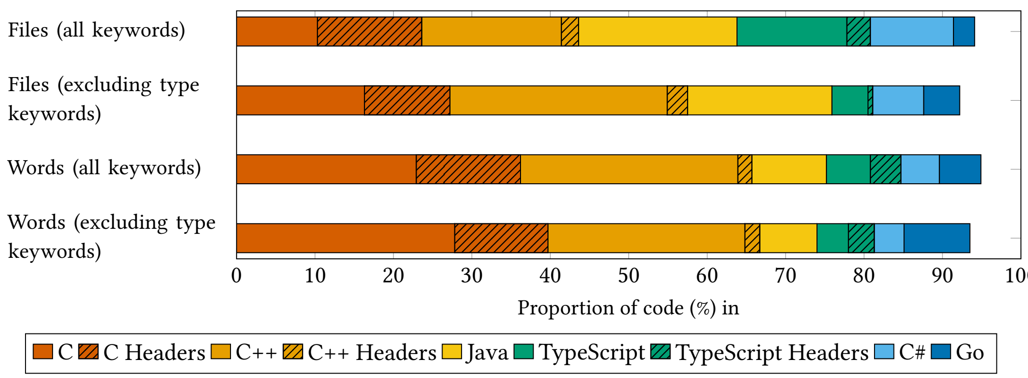

Fig. 3. Proportion of files and words in the dataset with floating-point keywords for the most used languages

can also be used in C++, and C++-specific headers (`.hpp, .hh, .hxx, .h++, .ipp, .tpp, .tcc`). Similarly, we treat TypeScript declaration files (`.d.ts`) separately from TypeScript source files (`.ts`).

We consider several metrics to identify which languages are most used for floating-point computations in our dataset. This comparison is illustrated in Figure 3. Each bar represents the proportion of files or words in the dataset that contain floating-point keywords, grouped by language. The first and third bars include all categories of keywords, while the others exclude keywords corresponding to floating-point types. We detail the choice of metrics and the corresponding results below.

Proportion of files with floating-point keywords (first bar): Counting the number of files, most of the code with any floating-point keyword is written in Java (20 %), while C++-specific headers are less common (2.2 %). The high number of Java files can be partly explained by the convention of isolating each class in its own file. In TypeScript, the use of the `number` type to represent both integers and floating-point values inflates the keyword count. To mitigate this effect, we also report the proportion of files excluding floating-point types and including only transcendental functions, arbitrary-precision and miscellaneous keywords.

Proportion of files with transcendental functions, arbitrary precision and miscellaneous keywords (second bar): In this case, C++ source files make up most of the code (28 %). Notably, the proportion of TypeScript drops to 4.6 % for source files and 0.62 % for declaration files.

Proportion of words of files with floating-point keywords (third bar): When counting words instead of files, the proportion of Java code drops as expected (9.5 %), and C and C++ source code has the largest share (28 % and 23 %, resp.).

From this observation we conclude that C, C++, Java, TypeScript, C# and Go are the most commonly used languages for floating-point computations in our dataset. We do *not* interpret this as evidence that these languages are inherently better suited for writing floating-point code, nor that, relative to their general usage, they are more frequently employed for writing numerical code than other languages. Regardless of the metric used, these six languages collectively account for more than 92 % of the dataset. We also explored other metrics, such as the number of lines of code and the exclusion of specific categories of keywords, but these variations did not lead to any substantial change in the overall trends.

5.3 Per-Function Analysis

Based on the observation of the previous section, we parse source files written in C, C++, Java, TypeScript, C# and Go and extract functions containing floating-point keywords (steps 8 and 9

Table 3. Statistics of extracted functions

Lang	# Files[a]	Files w/ fn[b]	Files w/ fn w/ kw[c]	# Fn w/ kw[d]
C	508 k	97 %	62 %	53 % ± 40 %
C++	878 k	99 %	81 %	41 % ± 33 %
C#	521 k	89 %	73 %	41 % ± 32 %
Go	134 k	95 %	70 %	31 % ± 31 %
Java	995 k	93 %	82 %	43 % ± 34 %
TS	690 k	58 %	71 %	48 % ± 33 %

[a] Number of files
[b] Proportion of files that have at least one function
[c] Proportion of files that have at least one function with a floating-point keyword (out of those with at least a function)
[d] Average proportion of functions with floating-point keywords per file (out of those with at least such a function), and standard deviation

in Figure 1). We again estimate the false-positive rate by manually inspecting a representative sample of 385 extracted functions. Among these, 368 functions (96 %) contain valid floating-point code. The remaining false positives fall into two categories: nine cases in which floating-point keywords are used as variable or library names, and eight functions that include floating-point types only in parsing-related code. Based on this sample, we estimate with 95 % confidence that the true positive rate of our extraction lies between 93 % and 98 %.

For each function, we count the number of lines of code, words, parameters, floating-point keywords of the different categories, loops, conditional statements, function calls and maximum nesting depth of these constructs. This is the main dataset we publicly release. Depending on the features a new verification tool is targeting, the developer can filter the dataset to sample relevant functions leveraging the above metrics.

Table 3 shows the number of files per language from which we extract functions (first column), and how many of these files contain at least one function (second column). The third column reports the proportion of those files that include at least one function with a floating-point keyword. The last column gives the average number of such functions per file (among files with at least one), along with the standard deviation. We observe that most languages have a high proportion of files with at least one function, except TypeScript (58 %), which includes many small files that export only constants. As previously described, most false positives in our manual inspection of files come from keywords in comments or string literals, but these are no longer present in our function dataset. The proportion of files with at least one function using a floating-point keyword aligns with the number of true positives we found manually during our per-file inspection. Due to the high variation in the data, we cannot draw a clear conclusion about the proportion of floating-point functions per file.

Answer to RQ1: Floating-point code is prevalent in open-source code. With 95 % confidence, we estimate that more than 62 % of projects contain floating-point code. Within projects, this code is often concentrated in a small number of files. We observe no clear trend in the proportion of functions using floating-point code within individual files.

Table 4. Function statistics per function. FPB = FPBench. Med denotes median and IQR the interquartile range, computed out of positive occurrences only.

Lang	Words		Params		Loops			Conditionals			Function calls			Keywords			
	med	IQR	med	IQR	> 0	med	IQR	> 0	med	IQR	> 0	med	IQR	Types	Trasc	Misc	MPFR
C	61	121	3	4	36 %	2	2	57 %	3	6	84 %	5	12	91 %	10 %	7.1 %	0.38 %
C++	42	94	2	2	22 %	2	2	44 %	3	5	76 %	6	14	95 %	6 %	5.8 %	0.06 %
C#	34	63	2	2	15 %	1	1	42 %	2	4	73 %	3	7	97 %	5.7 %	3.5 %	0.01 %
Go	42	97	2	2	29 %	2	2	63 %	2	4	85 %	6	14	97 %	3.4 %	5.2 %	0.48 %
Java	28	61	1	2	20 %	1	1	37 %	2	3	73 %	5	10	98 %	4.6 %	3.9 %	0.002 %
TS	27	43	1	1	15 %	1	1	44 %	2	3	83 %	3	6	98 %	1.6 %	4.0 %	—
FPB	23	26	2	2	12 %	1	0	6.9 %	1	1	61 %	2	1	100 %	38 %	0 %	0 %

(a) Floating-point code

Lang	Words		Params		Loops			Conditionals			Function calls		
	med	IQR	med	IQR	> 0	med	IQR	> 0	med	IQR	> 0	med	IQR
C	42	60	2	2	18 %	1	1	66 %	2	3	91 %	4	6
C++	26	40	2	2	13 %	1	1	37 %	2	3	81 %	3	7
C#	23	27	1	2	5.5 %	1	1	29 %	2	2	74 %	2	3
Go	28	45	2	2	15 %	1	1	54 %	2	2	80 %	4	8
Java	13	25	1	1	9.9 %	1	1	24 %	1	2	61 %	3	6
TS	17	28	1	2	6.8 %	1	0	33 %	1	2	76 %	2	4

(b) All code

Structure of Floating-Point Functions. Our final dataset consists of 9 945 253 functions, of which 1 168 217 are written in C (12 %), 4 176 635 in C++ (42 %), 2 574 341 in Java (26 %), 723 609 in TypeScript (7 %), 1 009 329 in C# (10 %), and 293 122 in Go (3.0 %). Table 4a reports on function complexity and the use of different language constructs in this dataset. For each construct, we report the proportion of functions that use it, along with the median and interquartile range (IQR, *i.e.,* the difference between the 75th and 25th percentiles) of its occurrences among those functions. For each keyword category, we report the proportion of functions using at least one keyword from that category. For illustration purposes, Figure 4 shows a randomly selected function with a miscellaneous keyword (`isFinite`); note that no other floating-point related keywords appear.

We note that the number of floating-point parameters and sizes of functions is relatively small, across all languages. C code tends to have the 'longest' functions. For each language, more functions contain function calls than conditional statements, and fewer contain loops.

In addition to the information in the table, we also analyse combinations of constructs: in C#, 10 % of functions include loops, conditionals, and function calls within a single function, compared to 26 % in C. When a function contains a loop, a conditional is also present in 69 % (Java) to 87 % (Go) of functions. Figure 5 is an example of a randomly selected C++ function from our dataset that contains both a loop and a conditional statement.

We compare these results with those obtained from non-numerical code. We uniformly sample 10 000 projects from our dataset and apply the same methodology, yielding 9 708 183 functions. Characteristics of these functions are summarised in Table 4b.

Compared to functions sampled from all projects, floating-point functions tend to be larger and contain loops more frequently. The larger average size of numerical functions may partly result

```go
func writeSVG(writer io.Writer, f biFunc) {
  fmt.Fprintf(writer, "<svg xmlns='http://www.w3.org/2000/svg' " +
    "style='stroke: grey; fill: white; stroke-width=0.7' " +
    "width='%d' height='%d'>\n", width, height)

  for i := 0; i < cells; i++ {
    for j := 0; j< cells; j++ {
      ax, ay := corner(i + 1, j, f); bx, by := corner(i, j, f)
      cx, cy := corner(i, j + 1, f); dx, dy := corner(i + 1, j + 1, f)
      if !isFinite(ax) || !isFinite(ay) || !isFinite(bx) || !isFinite(by) ||
          !isFinite(cx) || !isFinite(cy) || !isFinite(dx) || !isFinite(dy) {
        continue
      }
      r, g, b := color(i, j, f)
      fmt.Fprintf(writer, "<polygon points = '%g,%g,%g,%g,%g,%g,%g,%g'
       stroke = 'rgb(%d, %d, %d)'/>\n", ax, ay, bx, by, cx, cy, dx, dy, r, g, b)
    }
  }
}
```

Fig. 4. Randomly picked Go floating-point function in our dataset that contains miscellaneous keywords [3].

```cpp
void RolloutStorage::compute_returns(torch::Tensor next, bool use_gae, float gamma, float tau) {
    if (use_gae) {
        value_predictions[-1] = next;
        torch::Tensor gae = torch::zeros({rewards.size(1), 1}, torch::TensorOptions(device));
        for (int step = rewards.size(0) - 1; step >= 0; --step) {
            auto delta = rewards[step] + gamma * value_predictions[step + 1] * masks[step + 1] -
                 value_predictions[step];
            gae = delta + gamma * tau * masks[step + 1] * gae;
            returns[step] = gae + value_predictions[step];
        }
    }
    else {
        returns[-1] = next;
        for (int step = rewards.size(0) - 1; step >= 0; --step)
            returns[step] = returns[step + 1] * gamma * masks[step + 1] + rewards[step];
    }
}
```

Fig. 5. Randomly picked C++ floating-point function in our dataset that contains both a loop and a conditional statement [56].

from our keyword-based filtering, as larger functions are more likely to contain such keywords. The increased presence of loops could be a consequence of the larger average size of numerical functions. To assess whether function size alone accounts for the observed disparity in loop presence,

Table 5. Distribution of floating-point precision keywords used in functions per language: half (16-bit), single (32-bit), double (64-bit), quad (128-bit).

Module	**Precision**				**Mixed-Precision**
	Half	Single	Double	Quad	
C	2.7 %	41 %	52 %	1.9 %	6.5 %
C ++	0.1 %	60 %	38 %	0.7 %	3.9 %
C#	—	65 %	35 %	—	2.7 %
Go	—	23 %	82 %	—	8.5 %
Java	—	37 %	65 %	—	3.8 %

we perform a controlled analysis. We fit a binary logistic regression model [33] predicting the presence of a loop as a function of the logarithm of function size and whether the function contains floating-points. After controlling for size, numerical functions remain significantly more likely to contain loops, with coefficients ranging from 0.1215 ±0.003 for C++ to 0.6395 ±0.007 for C#, all with $p < 0.001$. According to the model, for functions of equal size, the odds of containing a loop are on average 13 % (C++) to 90 % (C#) higher in floating-point functions than in the general population. Although these results should be interpreted cautiously, they indicate that floating-point functions exhibit a systematically higher prevalence of loops beyond what can be explained by function size alone. The models achieve ROC AUC values between 0.79 (C) and 0.86 (Java). We do not observe a comparable effect for the presence of conditionals.

Mixed-Precision Code. We also analyse the use of different floating-point precisions within functions in our dataset. Table 5 reports, for each precision, the proportion of functions in which a corresponding type keyword appears, as well as the proportion of functions that use keywords from multiple precisions within the same function. We omit TypeScript from this analysis, as it provides only a single floating-point type (`number`). For C and C++, we additionally report half precision (16-bit) and quad precision (128-bit), which are supported by these languages.

Single (32-bit) and double (64-bit) precision are the most commonly used across all languages, athough their relative prevalence varies substantially by language. This variation may be partly explained by differences in the libraries commonly used in each ecosystem. For example, in C#, where 65 % of functions use single precision, the Unity library predominantly relies on 32-bit floating-point data types such as `Vector3` and `Quaternion`. In contrast, Go uses double-precision keywords in 82 % of functions, consistent with common practice in Go ecosystems where numerical libraries often operate on 64-bit floating-point values. Half (16-bit) and quad (128-bit) precision, as well as mixed-precision code, occur only rarely.

Parsing-Errors. Because not every source file is successfully parsed by Tree-sitter, we investigate how many files are affected by parsing errors and whether this biases our dataset toward parser-supported constructs. Tree-sitter recover from parsing errors by skipping invalid tokens and inserting placeholder nodes where syntactic elements are expected. Across the full dataset, Tree-sitter generate such error-recovery nodes in 16 % of the files. The languages most affected are C and C++, with 53 % and 33 % of files affected, respectively. Java and Go are significantly less affected, with 0.50 % and 0.23 % of files, respectively.

To assess the impact of these errors and characterise their causes, we manually inspect a sample of 97 files affected by parsing errors (corresponding to a Cochran sample size with a 10 % margin of error). In 90 out of 97 cases, parsing errors do not affect function extraction, *i.e.,* no functions are skipped. Based on this observation, we estimate with 95 % confidence that between 0.57 %

and 2.3 % of files in the full dataset have functions missed due to parsing errors. In most cases (74 out of 97), parsing errors stem from the use of preprocessor directives in C and C++ code. A common scenario involves conditional compilation directives (*e.g.,* `#ifdef`), which may introduce unmatched or duplicated closing braces at the end of a function. Because preprocessor directives can only be resolved with full knowledge of the surrounding code, they are difficult to be handled reliably when analyzing files in isolation. Other sources of parsing errors include macros used as keywords (*e.g.,* replacing `static` or `inline`) or macros that expand to language constructs such as `for` loops. We also observe cases in which Tree-sitter cannot correctly parse macro arguments without resolving macro expansion, for example when those arguments contain statements or type declarations. Finally, some errors (11 out of 97) arise from syntactic forms not supported by the Tree-sitter grammar, such as `new ptr*[n]`.

Answer to RQ2: In our dataset, conditional statements appear in many functions, up to 63 % in Go. This proportion increases when loops are present, although loops themselves are less frequent. For functions of equal size, floating-point functions are more likely to contain loops than functions sampled from all code. Function calls are also common in floating-point code, and typically these are not calls to transcendental functions. Special values like NaN or infinity appear rarely but are present. Despite these complexities, functions tend to be small, highlighting the importance of modularity in floating-point reasoning techniques. Single- and double-precision floating-point types are the most commonly used across all languages, whereas other precisions, arbitrary precision, and mixed-precision code are relatively rare.

Existing Benchmarks. We compare our dataset to FPBench [18], a benchmark suite of 131 floating-point functions written in Racket and designed for verification tools. FPBench functions can be exported to several supported languages. We export them to C and run our parser to extract statistics (see last row of Table 4a).

The FPBench functions have notably different characteristics: conditional statements and function calls appear less frequently, while calls to transcendental functions appear more often. The combination of a loop, conditional statement and a function call occurs only 1.5 % of the time (compared to 26 % in C). When a function contains a loop, a conditional is also present in 27 % of the cases (compared to over 69 % in real-world code). FPBench does align with our dataset in terms of the number of function parameters. The word count may be biased by the code being automatically exported to C from Racket and thus not written like a human would, *e.g.,* with fewer intermediate variables.

We presume that this difference arises because the FPBench benchmarks are drawn from existing literature and consist of hand-picked programs that current tools can already analyse successfully. In particular, many static floating-point reasoning tools offer limited support for conditionals, loops, and arbitrary function calls, which is reflected in the low prevalence of such constructs in FPBench.

The GNU Scientific Library (GSL) is also frequently used in the literature for benchmarking purposes. Guo et al. [31] use a subset of nine functions from the GSL Stats module. Other studies employ various subset of the GSL Special Functions module, ranging from 14 selected functions to the entire module [8, 46, 62, 64, 65].

We assess the prevalence of GSL usage in our dataset by computing the proportion of files and functions that use functions from the SF (special functions), Stats and Math (common mathematical functions) modules. We perform this analysis by supplying additional JSON files to the downloader and parser, following the same approach used for identifying floating-point keywords. For each module, we list all available functions and check for their presence in the dataset. This process

Table 6. Files and functions using GSL functions

Module	Files		Functions	
	C	C++	C	C++
GSL Math[a]	0.084 %	0.023 %	0.063 %	0.0073 %
GSL SF[b]	0.071 %	0.027 %	0.11 %	0.010 %
GSL Stats[c]	0.022 %	0.0034 %	0.013 %	0.0017 %

(Percentage out of files and functions with floating point keywords.)
[a] GNU Scientific Library common mathematical functions
[b] GNU Scientific Library special functions
[c] GNU Scientific Library statistics functions

also shows the flexibility of our implementation in supporting additional analyses on an existing dataset.

The results are summarised in Table 6. We observe that GSL modules are not widely used in the dataset. Only 0.084 % of C files and 0.023 % of C++ files with floating-point keywords include functions from the GSL Math module. The Stats and SF modules appear even less frequently. In comparison, the math.h library is included in 17 % of C and 14 % of C++ files with floating-point keywords. Usage at the function level is similarly sparse as only 0.13 % of C floating-point functions include calls to GSL SF functions.

Answer to RQ3: FPBench functions are representative of real-world floating-point code in terms of the number of parameters but not in terms of complexity. Conditionals are rare, and function calls, when present, typically involve transcendental functions that are over-represented. Functions from the GNU Scientific Library that have been used for evaluation appear rarely in our dataset.

5.4 Threats to Validity

A first threat to validity concerns the generalisability of our findings. By sampling data from GitHub and carefully filtering it, we analyse real-world code. However, industrial, closed-source software or open-source software from other sources than GitHub may exhibit characteristics different from those observed in open-source projects on GitHub. Large parts of our methodology and implementation are reusable on other data sources or within industrial case studies, which could help assess the extent of these differences in future work.

Another potential threat relates to the accuracy of metadata provided by GitHub. Commit timestamps depend on the local system time of the machine pushing the commit, which may be misconfigured. As a result, the recorded creation and last push dates of a project may not be reliable, which could affect our filter. A partial mitigation is to use the timestamp of the first commit instead of the reported project creation date, since an incorrect system clock would likely produce a consistent offset. However, this information is not included in the metadata returned by the API, and retrieving it explicitly would require an additional request per repository, effectively doubling the number of API calls. Similarly, GitHub's language classification is based on heuristics and may misidentify the main language of a project. This issue also applies to our own language detection, which relies on file extensions. Since some extensions are shared between multiple languages, misclassification is possible. Our manual inspection suggests, however, that this affects only a small fraction of the dataset.

A third threat relates to the keyword-based approach we use to detect floating-point code. Some transcendental functions appear as calls to external libraries, which we do not capture, potentially skewing our statistics. We also do not identify operations on complex numbers, which often rely on floating-point arithmetic. Abstractions such as type aliases, structures, and classes may also complicate detection of floating-point types, resulting in false negatives. We mitigate this by including a broader set of keywords, such as transcendental functions and special floating-point values, although this only partially addresses the issue. Keyword-based heuristics may also produce false positives. We reduce this risk by removing comments and ignoring string literals, as confirmed through manual inspection.

Duplicate removal represents another potential threat. Our approach eliminates token-level duplicates from the dataset, but does not remove near-duplicates, *i.e.,* files that differ only by a few characters or lines. Such near-duplicates can arise from different versions of the same library. Prior work by Lopes et al. [42] addresses this issue using token-similarity thresholds. However, it remains unclear how to define an appropriate threshold or how to assess its validity in our setting. We leave the investigation of near-duplicate detection for future work.

Finally, the tooling itself may contain bugs, which could introduce bias. We mitigate this risk through unit and integration testing. Additionally, when we perform uniform sampling, we manually verify that the resulting distribution is consistent with expectations.

6 Case Study: A Challenge Set of C Floating-Point Benchmarks

To demonstrate how our dataset can support practical benchmarking, we construct a set of 59 challenge benchmarks in C, which we upload along with the dataset. The choice to extract benchmarks in C is arbitrary (from the list of most frequently used languages for floating-point code). This case study is not intended as a definitive benchmark suite, but rather as an illustration of how our corpus can be used to generate realistic, ready-to-run benchmarks that reflect real-world floating-point practices. Each benchmark is a self-contained file containing a function extracted from our corpus together with all necessary dependencies to compile and run it independently.

Sampling and Extraction. The extraction process is semi-automated. We uniformly sample 200 functions from projects using permissive licenses (MIT, BSD, Apache-2.0). For each sampled function, we parse the source with Clang's Rust bindings[8] and recursively constructs its dependency graph by traversing the project's directory structure. Dependencies include type definitions and auxiliary functions. When a new file is discovered, we also store the include directives and macros it contains. When all dependencies are identified, we emit a single self-contained C file that aggregates the collected includes and macros, followed by the function and its dependencies in topological order. Because we do not rely on build systems such as `make`, extraction can fail in the presence of conditional compilation or cyclic dependencies (46 out of 200 sampled functions). Manual curation removes redundant dependencies and macros. To preserve self-containment and portability, we also exclude benchmarks that depend on external or platform-specific libraries (*e.g.,* OpenMP, CPython, `windows.h`), discarding 60 out of 155 remaining files. We further discard duplicates[9] (34 out of 94) and empty functions (1 out of 94), resulting in a final set of 59 unique benchmarks. We validate each extracted benchmark by compiling with both GCC 13.3.0 on Linux and Clang 21.1.8 on macOS.

Characteristics. The resulting programs span multiple domains, including scientific computing (44 %), embedded systems (12 %), software toolchains (12 %), graphics (10 %), and educational

[8]The extraction process is available as a subcommand in our framework.
[9]Although files in our dataset are deduplicated, identical function definitions may appear in distinct files, often due to code generation.

```c
double calSum(double data[], int N, int I) {
   if (I == 0) { return data[0]; }

   double sum = 0.f;
   int idx = 0;

   if (I >= N) { idx = N; }
   else { idx = I; }

   for (int i = 0; i < idx; i++) {
      sum += data[I - i];
   }

   return sum;
}
```

```c
void calAR(double PRICE[][4], int count, int N,
      double AR[]) {
   double hoValue[count];
   double olValue[count];

   for (int i = 0; i < count; i++) {
      hoValue[i] = PRICE[i][0] - PRICE[i][2];
      olValue[i] = PRICE[i][2] - PRICE[i][1];
   }

   for (int i = 0; i < count; i++) {
      double sum1 = calSum(hoValue, N, i);
      double sum2 = calSum(olValue, N, i);
      AR[i] = sum1 / sum2 * 100;
   }
}
```

Fig. 6. Example C floating-point benchmark extracted from our corpus. Unlike FPBench procedures, `calAR` is not self-contained and calls `calSum` [37].

material (10 %). Their size ranges from 4 to 1021 lines of code (LoC), with a median of 30 LoC (against 3 for FPBench). Unlike FPBench isolated numerical routines, the extracted functions preserve their original program context: several depend on auxiliary functions (29 %) or user-defined structures and unions (37 %), and most mix floating-point parameters with other types (66 %). Figure 6 illustrates one such benchmark, where the target function `calAR` calls an auxiliary routine `calSum` that computes the sum of the first `I` elements of an array. Benchmarks of this kind provide realistic cases for evaluating the interprocedural reasoning capabilities of floating-point analysis tools.

The extracted programs also differ from FPBench in the language constructs they employ. A large fraction use macros (31 %), pointer manipulation (63 %), and explicit type casts (41 %). They further exhibit floating-point usage patterns that extend beyond pure arithmetic or mathematical functions. For example, Figure 7 shows a benchmark performing low-level memory manipulation to encode the least significant bits of an input into a NaN payload, while other programs use floating-point variables as loop counters. These patterns illustrate how, due to C's permissive semantics, floating-point computations can appear in unexpected contexts. By exposing such patterns, these benchmarks prompt tool designers to make explicit and informed choices about which language features to support and which to explicitly exclude by design.

```c
static float quiet_nanf(float x) {
  uint32_t tmp;
  memcpy(&tmp, &x, 4);
  tmp |= 0x7fc00000lu;
  memcpy(&x, &tmp, 4);
  return x;
}
```

Fig. 7. C floating-point benchmark extracted from our corpus that uses low-level memory manipulation to store the least significant bits of the input into a NaN value [38].

By preserving real-world coding idioms that existing floating-point tools struggle to support, benchmarks extracted from our corpus will inevitably be challenging. We envision such benchmarks to serve as a challenging and realistic target for future tool development.

Table 7. Overview of representative freely available and recent floating-point reasoning tools. All listed tools support floating-point arithmetic.

Tool	input lang.	aim	supported features
Static tools			
FPTaylor [53]	custom	worst-case rounding error bounds	transcendental fncs.
PRECiSA [58]	PVS	worst-case rounding error bounds, mixed-precision tuning	transcendental fncs., map and fold over lists, non-recursive func. calls[a] conditional stmt, loops[a]
Daisy [4, 34]	Scala	worst-case rounding error bounds	transcendental fncs., some vector and matrix operations, non-recursive func. calls
Gappa [21]	custom	worst-case rounding error bounds	11 rounding modes (only arithm.)
Frama-C [10]	C	deductive program verification	essentially full C, limited support for rounding errors via Gappa
KeY [5]	Java	deductive program verification	subset of Java (roughly basic Java 1.2), no autom. support for rounding errors
Stainless [27]	Scala	deductive program verification	subset of Scala (mostly functional), no autom. support for rounding errors
Dynamic tools			
Herbie [47]	S-expressions	improve accuracy via rewriting	conditionals, fixed set of mathematical functions
Verrou [26]	integrated to Valgrind	numerical instability via Monte Carlo arithmetic	presumably all
Verificarlo [22]	C, C++, Fortran	numerical instability via Monte Carlo arithmetic	presumably all
FPLearner [60]	C/C++	mixed-precision tuning	presumably all
NSan [15]	LLVM	large rounding errors via shadow execution	presumably all

[a] Not evaluated systematically in publications as far as we know.

Applicability to Existing Tools. Table 7 provides an overview of a selection of existing tools. This list is not and is not meant to be exhaustive, rather we focus on available and recent tools that could potentially benefit from a benchmark suite such as the one that we extracted. (As stated previously, we do not intend our extracted C benchmarks as a definitive benchmark suite.) We focus on recent tools, i.e. tools whose code was updated in some way within the last 5 years, as older tools tend to be difficult to run in practice. The threshold of 5 years is arbitrary.

Many dynamic analysis tools target C/C++ in some way and so we expect that they can be immediately run on the extracted benchmarks. To use the tool Herbie, one would need to extract the numerical calculations out of the benchmarks.

To demonstrate that our benchmarks can be used in practice, we run NSan and Frama-C with Gappa as frontend on them. In contrast to other tools, NSan requires an explicit entry point with a `main` function. To accommodate this requirement, we add a simple `main` function to each benchmark that invokes the extracted function with fixed input values chosen to avoid runtime errors. Frama-C does not require an entry point and we therefore run it directly on the extracted benchmarks. NSan successfully analyses all the benchmarks and does not report precision-loss warnings (this is not very surprising, as such warnings are typically highly input dependent). Frama-C produces verification conditions for 54 out of the 59 benchmarks; the remaining five benchmarks use complex numbers, which are not supported by Frama-C.

For the remainder of the static analysis tools, one would need to translate them from C to their respective input language. We expect that as part of this rewrite, some of the currently unsupported features of each tool (*e.g.,* pointers) could also be expressed in a way that the tool can handle. This effort is beyond the scope of this paper.

An alternative option for these tools is to extract programs from GitHub directly written in the input language of interest. The newly added support for floating-point arithmetic in Stainless [27] successfully used Scala programs extracted in this code study for evaluation.

7 Discussion and Conclusions

Benchmarks used in the literature reflect the current capabilities of tools, rather than the way developers actually write code. Although function sizes are comparable, our study shows that the internal structure and feature usage of FPBench functions differ from user-written code. In particular, transcendental functions are overrepresented, whereas conditionals and their combinations with other control-flow constructs are significantly underrepresented. FPBench thus aligns more closely with the capabilities of existing static floating-point reasoning techniques, which provide limited support (if any) for conditionals and loops [10], and whose techniques are not modular. Our study further reveals that, despite being frequently used for benchmarking in the literature, calls to GSL functions appear rarely (less than 0.2 %) at both the file and function level.

We believe that our corpus can be used to extract code for different benchmark sets. Future research can also rely on this study to better gauge whether the complexity in a benchmark accurately reflects real-world floating-point usage.

To be effective on user-written code, floating-point analysis tools must prioritize modularity and reasoning about branching behaviour. Our study shows that floating-point code is typically modular, with most function bodies containing multiple calls to others. Conditionals are common, and loops almost always co-occur with conditionals. These characteristics suggest that the main challenge for analysis tools is not handling large, monolithic numerical kernels [20], but effectively analysing many small functions that combine modularity with non-trivial control flow. While tools such as PRECISA 4.0 [58] and Daisy [4] demonstrate promising approaches in this direction, the majority of state-of-the-art roundoff error analysers target self-contained code fragments.

Keyword-based approaches are unlikely to be effective for identifying floating-point code in dynamically typed languages. This study presents part of the big picture, and extending the analysis to dynamically typed languages would provide a natural complement. However, we believe this cannot be achieved by simply adapting our current methodology. Our findings show that in statically typed code, non-type keywords appear only rarely (6 %–17 % of functions depending on the language), and GSL-related keywords appear even less frequently, making non-standard math libraries an unreliable proxy. Without explicit type annotations, a keyword-based approach in dynamically typed languages would therefore raise serious concerns about the validity of the results. The presence of arithmetic operators would also be insufficient to identify floating-point usage, since they are commonly overloaded for integers or non-numerical data. Obtaining reliable type information in such languages would require instrumenting and executing the code with representative inputs, which in turn demands resolving dependencies and reproducing realistic

[10] Handling conditionals in static analyses of *e.g.,* rounding errors is challenging, because the real-valued specification and floating-point execution can take different paths through a program. Since worst-case rounding error bounds grow with every loop iteration, it is unclear how to find and prove loop invariants.

program executions (including GUI interactions where relevant). Designing methods to gather such information consistently and at scale remains an open challenge left for future work.

Semantically characterising floating-point code in a quantitative, scalable, and automated fashion raises substantial challenges. While it would be very interesting to identify a repository's application domain(s), detect common numerical algorithms, or assess numerical stability, these characterisations typically require manual annotation with multiple annotators to yield meaningful quantitative claims—we are not aware of automated tools that can do this reliably. Type-based analyses are similarly not as easily automated since they generally require successfully building projects. Prior work has attempted this for Java using build tools such as Maven or Gradle [32, 54, 55], but reported low success rates, which can bias the analysis. The process described in Section 6 to compile C benchmarks required several weeks of work for a single programming language and required manual intervention for a large number of projects. Developing a pipeline that addresses these issues is left for future work and can build on the baseline established by this paper.

8 Data-Availability Statement

All code, datasets, and artifacts necessary to reproduce the analyses and results presented in this paper are publicly available and permanently archived.

- *Scyros.* The source code of Scyros, our framework for large-scale code studies, is available on GitHub under the Apache 2.0 license at: https://github.com/fxpl/scyros. A versioned snapshot corresponding to the evaluated artifact is archived on Zenodo along with the artifact.
- *Function Dataset.* The corpus of metadata for floating-point functions extracted from GitHub, including links to the original functions, is archived on Zenodo under the Apache License 2.0 at: https://doi.org/10.5281/zenodo.17055622.
- *C Challenge Benchmarks:* the set of 59 self-contained C floating-point benchmarks extracted from our corpus is available on Zenodo as part of the function dataset.
- *Floating-Point Keywords List:* the list of keywords we used to identify floating-point code is also available on Zenodo as part of the function dataset.
- *Artifact.* The artifact accompanying this paper is a local web application that reproduces all analyses and results reported in the paper and provides a step-by-step guide for reproducing the study with different keywords on a small set of newly mined repositories. The artifact is archived on Zenodo under the Apache 2.0 license at https://doi.org/10.5281/zenodo.18500268 and has been submitted for Artifact Evaluation [28].

Acknowledgments

We thank the reviewers for their thoughtful feedback that strengthened our work. Additionally, we would like to thank Fridtjof Stoldt, Noé De Santo and Paulo Canelas for their suggestions and feedback on this work.

This work is co-funded by the European Union (ERC, HORNET, 101163629). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

This research was supported by the Swedish Research Council through the grant 2023-05161, "Scalable Error Verification: How Accurate is your Numerical Code?", and 2024-04565, "Data-race Freedom and Memory Safety for Untyped Languages".

References

[1] 2025. CodeQL. https://github.com/github/codeql
[2] 2025. Tree-Sitter. https://github.com/tree-sitter/tree-sitter
[3] 1-14j4z1-11. 2016. 1-14j4z1-11/Go: main.go. https://github.com/1-14j4z1-11/Go/blob/8af5ee9138a6993c1848e3e6892226c7339b2b61/ch03/ex03_03/src/main/main.go#L43
[4] Rosa Abbasi and Eva Darulova. 2023. Modular Optimization-Based Roundoff Error Analysis of Floating-Point Programs. In *International Static Analysis Symposium (SAS)*. doi:10.1007/978-3-031-44245-2_4
[5] Rosa Abbasi, Jonas Schiffl, Eva Darulova, Mattias Ulbrich, and Wolfgang Ahrendt. 2023. Combining Rule- and SMT-based Reasoning for Verifying Floating-Point Java Programs in KeY. *International Journal on Software Tools for Technology Transfer* 25, 2 (2023), 185–204. doi:10.1007/s10009-022-00691-x
[6] Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. 2014. Tracing Dynamic Features in Python Programs. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. doi:10.1145/2597073.2597103
[7] Edwin B. Wilson and. 1927. Probable Inference, the Law of Succession, and Statistical Inference. *J. Amer. Statist. Assoc.* 22, 158 (1927), 209–212. doi:10.1080/01621459.1927.10502953
[8] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic Detection of Floating-Point Exceptions. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/2429069.2429133
[9] Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. 2019. On the Impact of Programming Languages on Code Quality: A Reproduction Study. *ACM Trans. Program. Lang. Syst.* 41, 4 (2019), 21:1–21:24. doi:10.1145/3340571
[10] Sylvie Boldo and Claude Marché. 2011. Formal Verification of Numerical Programs: From C Annotated Programs to Mechanical Proofs. *Mathematics in Computer Science* 5, 4 (2011), 377–393. doi:10.1007/s11786-011-0099-9
[11] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129. doi:10.1016/j.jss.2018.09.016
[12] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. 2011. How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. doi:10.1145/1985441.1985448
[13] Casey Casalnuovo, Yagnik Suchak, Baishakhi Ray, and Cindy Rubio-González. 2017. GitcProc: A Tool for Processing and Classifying GitHub Commits. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. doi:10.1145/3092703.3098230
[14] William G. Cochran. 1977. *Sampling Techniques* (3 ed.). John Wiley & Sons.
[15] Clement Courbet. 2021. NSan: a Floating-Point Numerical Sanitizer. In *Proceedings of the International Conference on Compiler Construction (CC)*. doi:10.1145/3446804.3446848
[16] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *International Conference on Mining Software Repositories (MSR)*. doi:10.1109/MSR52588.2021.00074
[17] Ozren Dabić, Rosalia Tufano, and Gabriele Bavota. 2024. SEART Data Hub: Streamlining Large-Scale Source Code Mining and Pre-Processing. In *International Conference on Software Maintenance and Evolution (ICSME)*. doi:10.1109/ICSME58944.2024.00097 ISSN: 2576-3148.
[18] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. 2017. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In *Numerical Software Verification*. doi:10.1007/978-3-319-54292-8_6
[19] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. doi:10.1007/978-3-319-89960-2_15
[20] Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. 2020. Scalable yet Rigorous Floating-Point Error Analysis. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. doi:10.1109/SC41405.2020.00055
[21] Marc Daumas and Guillaume Melquiond. 2010. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Softw.* 37, 1 (2010), 2:1–2:20. doi:10.1145/1644001.1644003
[22] Christophe Denis, Pablo de Oliveira Castro, and Eric Petit. 2016. Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic. In *Symposium on Computer Arithmetic (ARITH)*. doi:10.1109/ARITH.2016.31
[23] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A Comprehensive Study of Real-World Numerical Bug Characteristics. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. doi:10.1109/ASE.2017.8115662
[24] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2015. Boa: Ultra-Large-Scale Software Repository and Source-Code Mining. *ACM Trans. Softw. Eng. Methodol.* 25, 1, Article 7 (2015). doi:10.1145/2803171
[25] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. In *Proceedings of the International Conference on Software*

Engineering (ICSE). doi:10.1145/2568225.2568295

[26] François Févotte and Bruno Lathuilière. 2017. Studying the Numerical Quality of an Industrial Computing Code: A Case Study on Code_aster. In *Numerical Software Verification*. doi:10.1007/978-3-319-63501-9_5

[27] Andrea Gilot, Axel Bergström, and Eva Darulova. 2026. Verifying Floating-Point Programs in Stainless. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[28] Andrea Gilot, Tobias Wrigstad, and Eva Darulova. 2026. Floating-Point Usage on GitHub: a Large-Scale Study of Statically Typed Languages (Artifact). doi:10.5281/zenodo.18500269

[29] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: Github's Data from a Firehose. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. doi:10.1109/MSR.2012.6224294

[30] Konstantin Grotov, Sergey Titov, Vladimir Sotnikov, Yaroslav Golubev, and Timofey Bryksin. 2022. A Large-Scale Comparison of Python Code in Jupyter Notebooks and Scripts. In *International Conference on Mining Software Repositories (MSR)*. doi:10.1145/3524842.3528447

[31] Hui Guo and Cindy Rubio-González. 2020. Efficient Generation of Error-Inducing Floating-Point Inputs via Symbolic Execution. In *Proceedings of the International Conference on Software Engineering (ICSE)*. doi:10.1145/3377811.3380359

[32] Foyzul Hassan, Shaikh Mostafa, Edmund S.L. Lam, and Xiaoyin Wang. 2017. Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. doi:10.1109/ESEM.2017.11

[33] David W. Hosmer, Stanley Lemeshow, and Rodney X. Sturdivant. 2013. *Applied Logistic Regression* (1 ed.). Wiley. doi:10.1002/9781118548387

[34] Anastasia Isychev and Eva Darulova. 2023. Scaling up Roundoff Analysis of Functional Data Structure Programs. In *International Static Analysis Symposium (SAS)*. doi:10.1007/978-3-031-44245-2_17

[35] Malin Källén and Tobias Wrigstad. 2021. Jupyter Notebooks on GitHub: Characteristics and Code Clones. *Art Sci. Eng. Program.* 5, 3 (2021), 15. doi:10.22152/PROGRAMMING-JOURNAL.ORG/2021/5/15

[36] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. doi:10.1145/2597073.2597074

[37] kayyyuan. 2018. kayyyuan/YKKline: YKIndicatorLib.c. https://github.com/kayyyuan/YKKline/blob/2476c8dc6f02e2e87427651c6edcb837b18ffab1/YKKline/YKKline/Model/IndicatorLib/YKIndicatorLib.c#L960

[38] keithw. 2024. keithw/wabt: fac.c. https://github.com/keithw/wabt/blob/b8d82acf6189a563830b0b76d89c4640c6310b33/wasm2c/examples/fac/fac.c#L423

[39] Ali M. Keshk and Robert Dyer. 2023. Method Chaining Redux: An Empirical Study of Method Chaining in Java, Kotlin, and Python. In *International Conference on Mining Software Repositories (MSR)*. doi:10.1109/MSR59073.2023.00080

[40] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. 2020. An Empirical Study on the Use and Misuse of Java 8 Streams. In *Fundamental Approaches to Software Engineering*. doi:10.1007/978-3-030-45234-6_5

[41] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F. Donaldson, Rafael Zahl, and Klaus Wehrle. 2017. Floating-Point Symbolic Execution: A Case Study in N-Version Programming. In *International Conference on Automated Software Engineering (ASE)*. doi:10.1109/ASE.2017.8115670

[42] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 84:1–84:28. doi:10.1145/3133908

[43] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. 2021. World of code: enabling a research workflow for mining and analyzing the universe of open source VCS data. *Empirical Softw. Eng.* 26, 2 (2021). doi:10.1007/s10664-020-09905-9

[44] Petr Maj, Stefanie Muroya, Konrad Siek, Luca Di Grazia, and Jan Vitek. 2024. The Fault in Our Stars: Designing Reproducible Large-scale Code Analysis Experiments. In *European Conference on Object-Oriented Programming (ECOOP)*. doi:10.4230/LIPIcs.ECOOP.2024.27

[45] Petr Maj, Konrad Siek, Alexander Kovalenko, and Jan Vitek. 2021. CodeDJ: Reproducible Queries over Large-Scale Software Repositories. In *European Conference on Object-Oriented Programming (ECOOP)*. doi:10.4230/LIPIcs.ECOOP.2021.6

[46] Dolores Miao, Ignacio Laguna, and Cindy Rubio-González. 2024. Input Range Generation for Compiler-Induced Numerical Inconsistencies. In *Proceedings of the International Conference on Supercomputing (ICS)*. doi:10.1145/3650200.3656618

[47] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/2737924.2737959

[48] Yun Peng, Yu Zhang, and Mingzhe Hu. 2021. An Empirical Study for Common Language Features Used in Python Projects. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. doi:10.

1109/SANER50967.2021.00012

[49] Paul Ralph, Sebastian Baltes, Domenico Bianculli, Yvonne Dittrich, Michael Felderer, Robert Feldt, Antonio Filieri, Carlo Alberto Furia, Daniel Graziotin, Pinjia He, Rashina Hoda, Natalia Juristo, Barbara A. Kitchenham, Romain Robbes, Daniel Méndez, Jefferson Seide Molléri, Diomidis Spinellis, Miroslaw Staron, Klaas-Jan Stol, Damian A. Tamburri, Marco Torchiano, Christoph Treude, Burak Turhan, and Sira Vegas. 2020. ACM SIGSOFT Empirical Standards. *CoRR* abs/2010.03525 (2020). doi:10.48550/arXiv.2010.03525

[50] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval that Men Do: A Large-scale Study of the Use of Eval in Javascript Applications. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. doi:10.1007/978-3-642-22655-7_4

[51] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/1806596.1806598

[52] Florian Sihler, Lukas Pietzschmann, Raphael Straub, Matthias Tichy, Andor Diera, and Abdelhalim Dahou. 2024. On the Anatomy of Real-World R Code for Static Analysis. In *International Conference on Mining Software Repositories (MSR)*. doi:10.1145/3643991.3644911

[53] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst.* 41, 1 (2018), 2:1–2:39. doi:10.1145/3230733

[54] Matúš Sulír and Jaroslav Porubän. 2016. A Quantitative Study of Java Software Buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. doi:10.1145/3001878.3001882

[55] Matúš Sulír, Michaela Bačíková, Matej Madeja, Sergej Chodarev, and Ján Juhár. 2020. Large-Scale Dataset of Local Java Software Build Results. *Data* 5, 3 (2020). doi:10.3390/data5030086

[56] TarrySingh. 2020. TarrySingh/cookbook: storage.cpp. https://github.com/TarrySingh/cookbook/blob/5a815b1429c9b3be3c4e192239488c141deeb00f/deep-learning/Deep-Reinforcement-Learning-Complete-Collection/PyTorch-cpp/src/storage.cpp#L59

[57] Ewan D. Tempero, James Noble, and Hayden Melton. 2008. How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software. In *European Conference Object-Oriented Programming (ECOOP)*. doi:10.1007/978-3-540-70592-5_28

[58] Laura Titolo, Mariano Moscato, Marco A. Feliu, Paolo Masci, and César A. Muñoz. 2025. Rigorous Floating-Point Round-Off Error Analysis in PRECiSA 4.0. In *Formal Methods (FM)*. doi:10.1007/978-3-031-71177-0_2

[59] Dominic Troppmann, Aurore Fass, and Cristian-Alexandru Staicu. 2024. Typed and Confused: Studying the Unexpected Dangers of Gradual Typing. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. doi:10.1145/3691620.3695549

[60] Yutong Wang and Cindy Rubio-González. 2024. Predicting Performance and Accuracy of Mixed-Precision Programs for Precision Tuning. In *Proceedings of the International Conference on Software Engineering (ICSE)*. doi:10.1145/3597503.3623338

[61] Yutong Wang and Cindy Rubio-González. 2024. Predicting Performance and Accuracy of Mixed-Precision Programs for Precision Tuning. In *Proceedings of the International Conference on Software Engineering (ICSE)*. doi:10.1145/3597503.3623338

[62] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient Automated Repair of High Floating-Point Errors in Numerical Libraries. *Proc. ACM Program. Lang.* 3, POPL (2019), 56:1–56:29. doi:10.1145/3290369

[63] Daming Zou, Yuchen Gu, Yuanfeng Shi, MingZhe Wang, Yingfei Xiong, and Zhendong Su. 2022. Oracle-Free Repair Synthesis for Floating-Point Programs. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 159:957–159:985. doi:10.1145/3563322

[64] Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. 2015. A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies. In *International Conference on Software Engineering (ICSE)*. doi:10.1109/ICSE.2015.70

[65] Daming Zou, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, Lu Zhang, and Zhendong Su. 2019. Detecting Floating-Point Errors via Atomic Conditions. *Proc. ACM Program. Lang.* 4, POPL (2019), 60:1–60:27. doi:10.1145/3371128

Received 2025-10-08; accepted 2026-02-17