On the Diversity and Frequency of Code Related to Mathematical Formulas in Real-World Java Projects

Oliver Moseler*[a], Felix Lemmer[a], Sebastian Baltes[a], Stephan Diehl[a]

[a] *Computer Science, University of Trier, Germany*

Abstract

In this paper, the term *formula code* refers to fragments of source code that implement a mathematical formula. We present empirical studies that analyze the diversity and frequency of formula code in open-source software projects. First, in an exploratory study, we investigated what kinds of formulas are implemented in real-world Java projects and derived syntactical patterns and constraints for reoccurring formula code fragments. We refined the patterns for sum and product formulas to not only automatically detect formula code in software archives, but also to reconstruct the implemented formula in mathematical notation. Second, in a quantitative study of a large sample of engineered Java projects on GitHub we analyzed the frequency of formula code and estimated that one of 700 lines of code in this sample implements a sum or product formula. We repeated the study with a sample consisting solely of scientific-computing projects, found a formula code density that was 7.4 times higher and estimated that one of 100 lines of code implements a sum or product formula. Our findings provide first insights into the characteristics of formula code, that can motivate further studies on the role of formula code in software projects and the design of formula-related tools.

Keywords: formula code, qualitative study, code patterns, GitHub, quantitative study

On the Diversity and Frequency of Code Related to Mathematical Formulas in Real-World Java Projects

1. Introduction

Since there exists a wide range of mathematical formulas and their implementations, in the context of this paper, we use the term *formula code* to denote fragments of source code that compute a numerical value (scalar, vector, matrix) in a way that can be expressed in a common mathematical notation [1] or by a mixed form of source code artifacts and maths symbols.

The correct and performant implementation of mathematical aspects is a crucial part influencing the success of a software system. The destruction of the Mariner 1 spacecraft in 1962 caused a $18.5 million financial damage due to a faulty implementation of a mathematical formula, in particular a missing superscript bar 'signifying a smoothing function, so the formula should have calculated the smoothed value of the time derivative of a radius' [2, 3, 4]. This is only one prominent example where formula code happens to be a critical part of a software system. The general discipline of software engineering is studded with problem solving and therefore with mathematical reasoning [5]. Thus maths is omnipresent in the work of a software engineer and represented in the software systems' code base by the utilization of one or more programming languages. Furthermore, the early detection of defects in maths implementations and discovery of opportunities to increase the software's performance within the development of a software system could save hours of testing and consumption of resources, especially in long running computational environments, such as scientific or high performance computing. Software development tools supporting the implementation and comprehension of formula code would not only be of advantage in such specialized scientific domains. To some extent every computer program contains at least some logic or discrete mathematics like combinatorics, probability theory, graph theory or number theory.

In psychological studies Landy et al. investigated the importance of spatial relationships in mathematical notations [6]. For example, test persons falsely rated the equation $a + b * c + d = c + d * a + b$ valid, if the distance between symbols did not correspond to the operator precedence. The authors concluded 'that competent symbolic reasoners typically rely on semantically irrelevant properties of notational formulae in order to quickly and accurately—but also sometimes inaccurately—solve symbolic reasoning problems.' Thus, We assume that a mathematical representation is beneficial in order to reduce both the time for comprehending the code as well as assessing its correctness. In mathematical notation, operators and symbols are arranged in two dimensions allowing a more compact view.

When we started this project, we found ourselves in the situation that we couldn't find much work to build on. Surprisingly, although mathematical formulas obviously play an important role in programming, so far, software engineering research has not empirically studied the implementation of formulas in common programming languages nor developed much tool support for implementing, debugging and optimizing formulas. Research as well as tools have rather focused on different levels of program abstractions (e.g. statement, class, or package level) or on higher-level abstractions (e.g. features, aspects, or components). In general purpose programming languages such as Java, Python or C++, formula code does not simply correspond to a syntactical category such as expression: Not all expressions in a program implement a formula (e.g. `fopen(fname)!=-1`) and not all formulas are implemented as expressions (e.g. program code for $\sum_{i=1}^{n} a_i$ typically uses `for`-loops). Identifying formula code and making software developers aware of recurring patterns, best practices, pitfalls, and useful 'hacks' related to the

implementation of mathematical aspects as well as simply showing the code in maths like notation can help to reduce development time and technical risk, as well as the effort for code comprehension and communication, particularly in cross disciplinary development teams, where experts of mathematically predominant domains, e.g. mathematicians or chemical scientists, and software developers tightly work together.

Our vision is that better understanding the characteristics of formula code will help to develop novel tools, beyond formula editors, for understanding, maintaining, debugging and optimizing formula code as well as to design new APIs and language features to enhance a software systems maintainability and thus its overall code quality.

To gain insights on the use of formula code in real software projects and as a basis for future research on tool support and language design, we want to answer the following two research questions:

RQ1 (Diversity): *What kinds of formula code occur in real-world software projects?*

RQ2 (Frequency): *How frequent is formula code both at file and line granularity?*

To answer these research questions, we performed two studies, one qualitative and one quantitative study. For our qualitative study, we ran a keyword-based search to find formula code in open source Java projects on GitHub (Section 3). While this keyword-based approach suffers both from low recall and low precision and requires a lot of manual post-processing, it helped us to find an initial set of real-world formula code samples, that we then manually analyzed in order to gain first insights on the kinds of formula code that exist (RQ1) and to derive patterns of formula code (Section 4). Given the diversity of the formulas that we found, we decided to focus on sum and product formulas for our quantitative analysis, because they are structurally non-trivial and we expected them to occur quite frequently. We will use the term *SP-formulas* in the rest of this paper to refer to sum and product formulas and the term *SP-formula code* to refer to source code we can express as SP-formulas in a mathematical notation. The derived patterns from the quantitative study form the basis of our pattern-based method for detecting SP-formula code (Section 5.1). Our approach is not only able to classify source code as SP-formula code, but also to reconstruct the formula implemented by the code in mathematical notation. Our evaluation shows that the pattern-based method has a very high precision (almost 100%) and a modest recall (31%) for SP-formula code. Moreover, for 85% of the detected SP-formula code the reconstructed formula was both correct and completely described the computation of the matched code. Finally, to answer research question RQ2, we applied the tool on a sample of 1000 different open source Java projects to detect SP-formula code on GitHub (Section 5). For a smaller sample, we also compared the densities from arbitrary application domains with those in scientific computing.

In summary our contributions are: A first qualitative study investigating the nature of formula code; a set of recurring formula code patterns derived from the findings in the qualitative study; a quantitative study on the density of sum and product formula code in open source Java projects.

2. Related Work

Work on automatic layout of formulas based on textual specifications as well as graphical formula editors [7, 8, 9] dates at least back to the seventies. There is also a considerable amount of research on producing internal representations from graphical ones using image recognition techniques [10, 11].

In the area of programming, recent work includes the integration of a formula editor in a common IDE as a domain specific language extension using Jetbrains MPS [12].

Since the advent of mining software repositories, researchers have developed numerous methods for analyzing software repositories to detect patterns in the source code (e.g., aspects[13] or API patterns [14]) or its changes (e.g., recommended changes [15] or refactorings [16]). The only work on reconstructing mathematical formulas from source code has been published by Moser et al. [17, 18]. Their RgB tool extracts formulas from annotated source code to produce a documentation of the source code. The tool was developed for Fortran and C++, requires manual annotations, uses static program analysis and covers only a small part of possible formulas. While there exist tools for searching mathematical formulas in documents (e.g., using tree-edit distance [19]) or for searching mathematical expressions in software binaries [20] (using fingerprints) in order to build a search system which is capable of querying for software libraries implementing a mathematical term, to our knowledge, so far, no methods for detecting formula code in software repositories or empirical studies on formula code in common programming languages have been published. To some extent, almost every implementation of a mathematical formula is working with numerical values such as integer or floating point numbers. The computation of numerical values comes with its own challenges and pitfalls. Recently, studies on characteristics such as categories, symptoms, frequencies and possible fixes of numerical bugs have been conducted by Di Franco et al. [21]. They have neither investigated patterns in the code of numerical computations nor tried to detect it automatically, but focused on bugs which are related to these code fragments. While parts of numerical computations can be described by formulas, they are often more algorithmic in nature.

3. Keyword-Based Search for Formula Code

To get first insights into what kinds of formulas are actually implemented in real-world software projects (RQ1) we conducted a qualitative study using GitHub as our data source. With more than 96 million repositories and 31 million contributors involved (as of September 2018 [22]), GitHub is one of the most popular code hosting platforms today. It is not only used by developers for their personal projects, but also by large companies such as Google, Microsoft, or Facebook. We restricted our search to projects containing Java source code, since Java is one of the most popular general purpose programming languages according to the IEEE language ranking (as of July 2018 [23]) and the TIOBE index (as of October 2018 [24]). Our decision to investigate Java projects is based on the assumption that they contain code that was developed by programmers of all levels from novices to experts with respect to their programming as well as maths skills.

To find formula code, we searched for certain keywords in the commit messages and code comments, because software developers use these annotations to document and communicate various aspects of source code artifacts and changes, including their intent. For our study, we wanted to find program code that a developer documented to be associated with mathematical formulas. Hence, we searched for occurrences of the keywords *'formula'*, *'equation'*, *'math'*, *'theorem'*, *'sum of'* and *'product over/of'* within the Git commit logs and comment sections of Java code . While our list of keywords is certainly not comprehensive, it allowed us to find a sufficiently large and divers set of samples for our subsequent manual analysis.

In a first attempt, we utilized the Boa language and infrastructure [25] to retrieve candidates. We conducted a search on the '2015 September/GitHub' dataset available through Boa. This approach revealed only a modest number of useful hits. Furthermore it was a time consuming

task to deduce the interesting code fragments, if existent, from the commit log messages since a commit usually relates to multiple files and thus required a significant amount of manual effort to identify the relevant file and source-code fragment in this file. Thus we changed our strategy and only searched in the source code comments, because the comments are in the same file and usually very close to the source code fragments that they annotate. Unfortunately, Boa did not provide access to source code comments, since they were not contained in its data model. As a consequence, we switched to Google Big Query [26] (GBQ), a web service for searching GitHub using the GBQ GitHub and GHTorrent datasets, that allows executing SQL-queries to search for keywords within source code comments of Java files.

3.1. Exploratory study

With the help of GBQ, we created one big CSV file containing all matches of each keyword mentioned above. Each match is recorded as a tuple with the following data:

```
(id, match, link, line, repo_name, path)
```

where `id` denotes a file identifier in the GBQ GitHub dataset, `match` the matched keyword, `link` the link to the respective repository on GitHub, `repo_name` the corresponding repository name consisting of the account as well as project name and `path` denotes the path of the file containing the match.

The goal of our qualitative analysis was to find common properties of real-world formula code. To this end, we followed an iterative analysis process borrowing some ideas (open coding, iteration, theoretical sampling and saturation) from Grounded Theory [27]:

1. Compute a set of matches (sample) using feedback from the previous iteration to adapt the sampling strategy and collect data that will more likely lead to new insights (theoretical sampling).
2. For each match:
   - Decide whether the source code fragment implements a formula
   - Try to reconstruct the underlying mathematical formula, observe and describe phenomena of the formula and the formula code (open coding).
3. If the analysis of the sample lead to new insights (i.e. new or refined codes) then repeat the analysis with another sample (Step 1). Otherwise, our codes cover all relevant phenomena (theoretical saturation).

In total, we conducted 4 iterations of group sessions until we reached a saturation on our observations. For these iterations, we used the following samples to search for formula code out of different perspectives:

Top10M: Matches of top 10 projects sorted by number of matches per project (keywords: theorem, formula, math, equation, sum of)

Top50P1: Matches of top 50 projects sorted by popularity (keywords: theorem, formula, equation, sum of)

Top20P2: Matches of top 20 projects sorted by popularity (keywords: product of, product over)

Figure 1: Formula reconstructed from code in Listing 2

Random: Matches randomly selected (keywords: theorem, formula, equation, sum of, product of, product over)

For the first sample, we ranked the projects by the number of matches and started to manually inspect the matches of the top ten projects. As the term *math* caused too many false positives, i.e. mostly commented calls to functions of the standard library like `Math.sqrt(...)`, we omitted it in subsequent samples. The samples TOP50P1 and TOP20P2 were created by sorting the projects by their popularity, measured in terms of their number of watchers. Our motivation was to reduce noise in our sample caused by small toy projects.

In total, we closely inspected 142 matches from 101 different open source Java projects on GitHub and found 47 matches from 21 different projects to be formula code. For those matches, we reconstructed the underlying mathematical formula of each fragment (example shown in Figure 1). The observed and coded phenomena include control flow statements, roles of variables, comprehensibility and mathematical data type. Note, albeit we consider the Java language feature of lambda expressions being close to a mathematical notation, it did not occur once in our qualitative study.

Control-Flow Statements. As shown in Table 1, 24 of the 47 verified formula code fragments made use of a `for`-loop of which 12 were simple non nested `for`-loops, eight were double nested `for`-loops (one nested into another) and four were triple nested `for`-loops. We only found two examples where `while`-loops were used to implement a mathematical formula—one was non nested and the other double nested. Furthermore, our findings revealed that most sum and product formulas were implemented using `for`-loops, which is not surprising as `for`-loops are closer to the mathematical representation than other loops. The same holds for formulas which required iterating through a vector or matrix. The conditional statement (`if`) was found 16 times, either within a loop body to function as a filter (seven times) or completely outside of any loop (nine times). In three cases, we also determined an incremental computation which means that, at runtime, a series of method calls resulted in a more complex implementation of a mathematical formula—the formula actually describes the invariant of the value of a variable. For example, the method `put(float value)` in the class `FloatCounter` of the project `libgdx/libgdx` (Listing 1) incrementally updates simple statistical values, such that `average` equals $\frac{1}{n}\sum_{i=1}^{n} v_i$ where n is the current value of `count` and v_i are the actual values of the parameter `value` of all invocations of the method:

Listing 1: Formula code in `FloatCounter.java` [28]

```
public void put (float value) {
    latest = value;
    total += value;
    count++;
```

Table 1: Properties of the coded samples

		control-flow statements				roles of variables				comprehensibility				mathematical data type			
Sample	coded	**for**	**while**	**if**	incremental	read only	accu	index	co-index	w/o context	expressible	proximity	mix-form	matrix	array	series	point/vector
Top10M	12	4	1	5	1	9	3	5	0	7	11	9	3	4	9	2	5
Top50P1	21	10	1	8	2	17	12	13	2	11	18	11	8	5	7	8	8
Top20P2	4	3	0	0	0	4	3	3	0	3	4	4	1	1	3	3	3
Random	10	7	0	3	0	9	4	7	1	7	10	7	6	2	4	2	6
Total	47	24	2	16	3	39	22	28	3	28	43	31	18	12	23	15	22

```
    average = total / count;
    ...
```

Albeit recursive implementations are considered to be more elegant and closer to the mathematical specification, we found no recursively implemented mathematical formula in our sample. This may be due to the fact that recursive implementations suffer from performance penalties. Moreover, Grechanik et al. [29] analyzed 30.000 Java projects on SourceForge and found that less than 4% of all methods were recursive.

Roles of Variables. We categorized variables by the way they are initialized, read and changed in the formula code—i.e. the roles they play in the code. Table 1 lists the different roles of variables that we found in our examples. These different kinds of variable roles mostly refer to their use within a loop. First of all, it makes a difference whether a variable is read or written in the code. If a variable is only read, it is usually a parameter of a method, a field or a constant. More interesting are the write-accessed variables. Here, we distinguish accumulator variables, indexing variables and co-indexing variables. Accumulator variables are used to accumulate a value over every iteration of a loop and occur on the left side of an assignment expression. Indexing variables are usually incremented in each iteration of a loop and are mainly used to access data structures like arrays or collections, or used in an expression to generate a series of values. In `for`-loops, the indexing variable is usually explicitly defined in the head of the loop.

A co-indexing variable, like the name suggests, is a variable that indirectly depends on the value of the current indexing variable and is possibly used in expressions in each iteration of the loop. For example, the variable `pixel` in the method `analyze(BufferedImage image)` of the class `HOGFeature.java` in the project `airbnb/aerosolve` is a co-index of the index variable `i` (see Listing 2):

Listing 2: SP-Formula code in `HOGFeature.java` [30]

```
// Compute sum of all channels per pixel
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    int pixel = image.getRGB(x, y);
    for (int i = 0; i < 3; i++) {
```

```
        lum[x][y] += pixel & 0xff;
        pixel = pixel >> 8;
      }
      lum[x][y] /= 3;
} }
```

We only found few instances of co-indexing variables in our coded examples. Finally, we also found temporary variables, which were used to split the computation of an expression into several steps. We assume that temporary variables were often used to increase readability and to support debugging.

Roles of variables have also been investigated in computer science education research [31]. Our roles of variables correspond to the ones identified by Taherkhani et al. [32] in their investigation of implementations of sorting algorithms. Our term *indexing variable* relates to their definition of a *stepper*, our term *co-indexing variable* to their definition of a *follower*, and our term *accumulation variable* to their understanding of a *gatherer*. Finally, they denote constants or read-only variables as *fixed values*.

Often, valuable context information was encoded in the name of a variable, e.g. `C_phi` (C_ϕ) or `xbar` ($\overline{x}$). We assume that developers name variables or other artifacts like this when they program according to a visual representation of a mathematical formula. This way, a mental and partly visual connection between the program code and the underlying mathematical representation is established to increase the recognition value of the formula within the code.

Comprehensibility. The effort for reconstructing the formulas was influenced by several factors (see Table 1). In many cases, the formulas could be reconstructed by inspecting the code fragment and without additional context information. In other cases, we had to inspect the code surrounding the actual denoted formula, because the code of the complete implementation can be fragmented in many lines of code, methods or even classes.
Furthermore, we recorded whether we were able to express the documented formula code in a mathematical notation (row *expressible* in Table 1). The few cases for which we could not reconstruct the underlying formula, but which were annotated by the developers to be implementations of some formula, are either indications of our limited domain knowledge or examples of how performance improvements obfuscate the program code.
A related property of the formula code is its proximity to the mathematical representation. With 33 of 47 coded matches, the majority of the inspected formula code examples are quite close to their mathematical notation. When reconstructing the formulas, we often caught ourselves mixing mathematical notations with simple program code artifacts, e.g. function calls or array accesses. Therefore, we also coded whether a code fragment could be more intuitively described by a mixed representation of mathematical symbols and program code notation, even if we were able to reconstruct a purely mathematical representation as well. Figure 1 shows an example where we embedded code fragments into the mathematical representation since the bit operations `&` and `>>` are programming specific and have no corresponding semantically equal symbols in maths. Altogether, for 18 of 47 coded examples, we found that such a mixed form might be an appropriate alternative representation.

Mathematical Data Type. Twelve of the 47 formula code examples implement matrix operations or calculations on matrices in general. Usually, matrices were implemented with the help of two-dimensional arrays either directly or indirectly through utility classes. But simply assuming a formula code example is implementing matrix calculations when detecting a two dimensional

array is not always correct. Among other things, the correct sizes of the dimensions were necessary to consider a two dimensional array to be a matrix implementation. Furthermore, a two dimensional array in conjunction with a nested for-loop may not have been intended to be an actual matrix operation even though one could express it as such. Almost twice as many formula code examples (22 of 47) dealt with vector calculations or points in 2D or 3D space. In the coded examples, vector mathematics was implemented in three different ways: by arrays (e.g. `v[0]`), by separate scalar variables for each dimension of a vector (e.g. `v_x`, mostly for 2D or 3D vectors), or by utility classes (e.g. `v.get(0)`).
Finally, 15 of 47 examples implemented a mathematical series with a single loop. In mathematics, a series is an *infinite* sum which of course is not implemented as such and thus outlines an important difference between program code representation and mathematical notation of the same formula.

4. Derived Formula Code Patterns

In our exploratory study, we found many different structural aspects and other phenomena related to formula code. We also gained more insights in what developers annotated to be program code implementing a mathematical formula. Moreover, we were also able to derive patterns of typical formula code examples from our observations. In this section we will explain these in detail.

Since more than half of the examples in our qualitative study used `for`-loops, we focus on loop-related patterns in the following. In particular, we look at those using `for` or `foreach` to implement a sum or product formula. In this section, we exemplary present a simple (non nested) `for`-loop and a nested `foreach`-loop pattern in an abstract notation including derived constraints for a transformation of the code to a formula representation in mathematical notation. Please note that arithmetic operations and function calls may occur in expressions within these formulas. All patterns that we implemented in our detection tool to perform the case study in Section 5 can be described in the same manner.

4.1. Non-nested for loops

A pattern of typical implementations of sum and product formulas using a `for` loop is presented in Pattern 1:

Pattern 1: Syntactic pattern for implementations of sum/product formulas using a `for`-loop

```
for ( index = exp1;
      index < exp2;
      [ index=index+1 | index+=1 | ++index | index++ ] ) {
        block1
        [ accu = accu op exp3; | accu op= exp3; ]
        block2
}
```

In this pattern, we used the variable roles introduced in the previous section to name the meta variables. The meta variables $expr_i$ are placeholders for arbitrary expressions, $block_i$ for any possible other program code at those positions. This implies that the code implementing a formula can be embedded in other program code and that only a small slice of code forms the actual

formula code that we can represent in mathematical notation. In the above pattern, we increment the indexing variable by one in each iteration and make use of the comparison operator *less than* in the loop conditional. This is due to the fact that it is the most common utilization of a `for`-loop in this context, and a stepping of one is consistent with the semantics of a mathematical sum or product. Other loop-conditions or increments are conceivable but would result in a more complex mathematical representation albeit some, e.g. using the comparison operator *less than or equal*, are trivial adjustments. In the patterns that we implemented, we therefore allowed other relational operators as well. Assuming that the value of the variable *accu* is $accu_0$ before the first execution of the `for`-loop, the mathematical formula that relates to the formula code pattern in Pattern 1 is described by the following formula template:

$$accu = accu_0 \;\; op \sum_{index=exp_1}^{exp_2-1} exp_3 \quad \text{if } op \in \{+, -\} \tag{1}$$

$$accu = accu_0 \;\; * \prod_{index=exp_1}^{exp_2-1} exp_3 \quad \text{if } op \in \{*\} \tag{2}$$

$$accu = accu_0 \;\; * \prod_{index=exp_1}^{exp_2-1} \frac{1}{exp_3} \quad \text{if } op \in \{/\} \tag{3}$$

However, many program code fragments that match with the syntactic pattern may not implement the formula described by the formula template, e.g. because they change the accumulator or indexing variables in the code blocks or expressions. Hence, we extended the pattern with additional constraints to reduce the number of false positives. To this end, we define the functions *vars*() and *writes*() which we will later use to formulate the constraints.

$$\mathbf{vars}(e) = \{x \mid x \text{ occurs in } e\} \tag{4}$$

$$\mathbf{vars}(\{e_1, \ldots, e_k\}) = \mathbf{vars}(e_1) \cup \cdots \cup \mathbf{vars}(e_k) \tag{5}$$

$$\mathbf{writes}(b) = \{v \mid v{=}e \text{ occurs in } b\} \tag{6}$$

$$\mathbf{writes}(\{b_1, \ldots, b_k\}) = \mathbf{writes}(b_1) \cup \cdots \cup \mathbf{writes}(b_k) \tag{7}$$

The first two constraints require that the accumulation variable *accu* shall not occur in the expressions $expr_2$ and $expr_3$ and the indexing variable *index* shall not occur in $expr_2$:

$$accu \notin \mathbf{vars}(\{exp_2, exp_3) \tag{8}$$

$$index \notin \mathbf{vars}(exp_2) \tag{9}$$

We allow that the accumulation variable occurs in $expr_1$, because it will only be evaluated once during the initialization of the indexing variable and at that time it will have its initial value $accu_0$. It must not occur in $expr_2$, because then it would occur on both sides of the equal sign in the formula. The indexing variable must not occur in $expr_2$, because otherwise the upper bound of the sum or product would not be constant but reevaluated in each iteration.

The following three constraints require that any variable occurring in $expr_2$ and $expr_3$ as well as the variables *accu* and *index* will not be written within the body of the loop, i.e. that there are

no assignment statements assigning new values to these variables:

$$accu \notin \textbf{writes}(\{block_1, block_2\}) \tag{10}$$

$$index \notin \textbf{writes}(\{block_1, block_2, exp_3\}) \tag{11}$$

$$\textbf{vars}(\{exp_2, exp_3\}) \cap \textbf{writes}(\{block_1, block_2, exp_2, exp_3\}) = \emptyset \tag{12}$$

Variables can not only be changed directly through assignments, but also indirectly through method calls in the body of the loop. In this case, the method has a side effect: Rountev [33] describes a side effect of a method as '(...) state changes that can be observed by code that invokes the method'. Thus, if all methods called in the body of the loop are side-effect free, we can be certain that they don't change the relevant variable values. We did not add a constraint on side-effect-freeness, because it would require a full-fledged static program analysis.

4.2. Nested loops

We also defined patterns for nested loops, loops which iterate over arrays, and `foreach`-loops which iterate over collections (including arrays). As an example, we present a pattern with two nested `foreach`-loops below:

Pattern 2: Syntactic pattern for implementations of computing a vector of sums/products using two nested `foreach`-loops

```
for( entry : exp1 ) {
    block1
    for( elem : exp2 ) {
        block2
        [ entry = entry op exp3 | entry op= exp3 ]
        block3
    }
    block4
}
```

In the formula template related to this pattern, we use the mathematical notation for an indexed family:

$$\left(entry \;\; op \sum_{elem \in exp_2} exp_3 \right)_{entry \in exp_1} \quad \text{if } op \in \{+, -\} \tag{13}$$

$$\left(entry * \prod_{elem \in exp_2} exp_3 \right)_{entry \in exp_1} \quad \text{if } op \in \{*\} \tag{14}$$

$$\left(entry * \prod_{elem \in exp_2} \frac{1}{exp_3} \right)_{entry \in exp_1} \quad \text{if } op \in \{/\} \tag{15}$$

To reduce the number of false positives we define the following constraints for the nested `foreach` pattern:

$$entry \notin \mathbf{writes}(\bigcup_{i=1}^{4} block_i) \tag{16}$$

$$elem \notin \mathbf{writes}(\{block_2, block_3\}) \tag{17}$$

$$\mathbf{vars}(exp_1) \cap \mathbf{writes}(\bigcup_{i=1}^{4} block_i) = \emptyset \tag{18}$$

$$\mathbf{vars}(\{exp_2, exp_3\}) \cap \mathbf{writes}(\{block_2, block_3, exp_3\}) = \emptyset \tag{19}$$

4.3. Vector arithmetics

Although we did not use these patterns in our later analysis, we also defined patterns for vector addition and scalar product. We present a pattern for the 2D vector space which can easily be extended to more dimensions. Note that Pattern 1 would apply for scalar products as well if they are implemented using a `for`-loop. Furthermore, we did not observe any vector addition with more than three dimensions being implemented in the manner of the following pattern.

Pattern 3: Syntactic patterns for implementations of vector addition and scalar product

`Scalar product:`
var `=` $exp_{1,1}$ `*` $exp_{2,1}$ `+` $exp_{1,2}$ `*` $exp_{2,2}$ `;`

`Vector addition:`
var_1 `=` $exp_{1,1}$ `+` $exp_{2,1}$ `;`
var_2 `=` $exp_{1,2}$ `+` $exp_{2,2}$ `;`

The formula template for these patterns uses typical vector notation:

$$var = \left\langle \begin{pmatrix} exp_{1,1} \\ exp_{1,2} \end{pmatrix}, \begin{pmatrix} exp_{2,1} \\ exp_{2,2} \end{pmatrix} \right\rangle \quad \text{and} \quad \begin{pmatrix} var_1 \\ var_2 \end{pmatrix} = \begin{pmatrix} exp_{1,1} \\ exp_{1,2} \end{pmatrix} + \begin{pmatrix} exp_{2,1} \\ exp_{2,2} \end{pmatrix} \tag{20}$$

The above patterns would match with far too many assignments in the source code, thus we add the following constraints. Our constraints are based on the observation that certain suffixes occur often: `.x`; `getX()` or `[0]` and that certain naming conventions are used, e.g. `sx` or `s1`, to access components of a vector. First, we define the following auxiliary functions:

$$\mathsf{source}(e) = e', \text{if } e = e'.s \text{ and } s \in \{\texttt{x}, \texttt{y}, \texttt{getX}(), \texttt{getY}(), \texttt{get}(\texttt{0}), \texttt{get}(1)\} \text{ or } e = e'[i] \tag{21}$$

$$\mathsf{source}(e) = p, \text{if } e = ps \text{ is a variable name with prefix } p \text{ and suffix } s \in \{\texttt{x}, \texttt{y}, 0, 1\} \tag{22}$$

$$\mathsf{index}(e) = 0, \quad \text{if } e = e'.s \text{ and } s \in \{\texttt{x}, \texttt{getX}(), \texttt{get}(\texttt{0})\} \text{ or } e = e'[0] \text{ or } e \text{ is a variable name with suffix } s \in \{\texttt{x}, 0\} \tag{23}$$

$$\mathsf{index}(e) = 1, \quad \text{if } e = e'.s \text{ and } s \in \{\texttt{y}, \texttt{getY}(), \texttt{get}(1)\} \text{ or } e = e'[1] \text{ or } e \text{ is a variable name with suffix } s \in \{\texttt{y}, 1\} \tag{24}$$

Now we can define the following constraints:

$$\mathsf{source}(e_{1,1}) = \mathsf{source}(e_{1,2}) \text{ and } \mathsf{source}(e_{2,1}) = \mathsf{source}(e_{2,2}) \tag{25}$$

$$\mathsf{index}(e_{1,1}) = \mathsf{index}(e_{2,1}) \text{ and } \mathsf{index}(e_{1,2}) = \mathsf{index}(e_{2,2}) \tag{26}$$

For the vector addition we also require:

$$\mathsf{index}(var_1) = \mathsf{index}(e_{1,1}) = \mathsf{index}(e_{2,1}) \tag{27}$$

$$\mathsf{index}(var_2) = \mathsf{index}(e_{1,2}) = \mathsf{index}(e_{2,2}) \tag{28}$$

$$\mathsf{source}(var_1) = \mathsf{source}(var_2) \tag{29}$$

Now the formula templates can be rewritten as:

$$var = \langle\, \mathsf{source}(exp_{1,1}), \mathsf{source}(exp_{2,1}) \,\rangle \tag{30}$$

$$\mathsf{source}(var_1) = \mathsf{source}(exp_{1,1}) + \mathsf{source}(exp_{2,1}) \tag{31}$$

Note that the presented patterns with the given constraints are only a heuristics to be able to find candidates for formula code, i.e. instances of the pattern within Java software projects. The patterns neither define necessary nor sufficient conditions of program code implementing, for example, a sum or product formula.

5. SP-Formula Code on GitHub

To answer research question RQ2, we performed a quantitative study on two different samples of open source Java projects on GitHub. To automate the search for formula code, in particular SP-formula code, we developed a detection tool which employs refined variations of the patterns introduced in Section 4. First, we present some detail on the SP-formula code detection tool, in particular on the patterns it can detect and its evaluation in terms of precision and recall. Next, we introduce a sample of engineered Java software projects of arbitrary topic (see GitHub *topics* [34]) and present the results obtained by our tool for this sample. We also look at the application domains of the projects with highest SP-formula code densities. Thereafter, we describe another sample consisting solely of Java projects with topic *scientific-computing* and discuss the results computed by our tool for this sample.

5.1. Pattern-Based SP-Formula Code Detection Tool

For our study, we used a shell script that clones each project from a given list of GitHub projects and checks out their configured default branch. The detection tool reads all Java files of these projects one after another (comments in the source code are removed) and searches for all patterns in parallel to exploit multiple CPU cores. Each pattern is implemented as a compiled regular expression. Although the regular expressions cannot match every syntactic pattern of the Java programming language, the approach performed pretty well for our purposes as it also captures and preprocesses pattern-specific elements like variable roles, such that the constraints associated with each pattern can be tested. The tool tests the nested SP-formula code patterns first and then the non-nested SP-Formula-code patterns (see Table 2). Every match of the non-nested patterns will be checked for intersection with all matches of the nested patterns. If an intersection exists, the respective non-nested match will be discarded. Matched code fragments that satisfy the constraints are attached to the output which is stored in form of a CSV file. This file lists for each entry, the line numbers of the start and end of the match, the source code of the code fragments, the inferred formula in mathematical notation as well as other detailed information about the source like project name, file name and GitHub path. The inferred formula is basically an instance of our formula templates and is stored in the file both in a textual representation as well as in MathML.

Listing 3 shows the regular expression for Pattern 1 (all pattern implementations are available in the supplementary material [35]). Note that the regular expression is built using String constants like `VAR` and `EXP` which themselves contain regular expressions:

Listing 3: Regular expression for non-nested `for`-loop implementing a sum/procuct (Pattern 1)

```
Pattern.compile(
  // head of loop
  "(?<lineOuterFor>for"+b+"\\((?:"+b+DT+b2+")?"+
  "(?<ind0>"+VAR+")"+b+"="+b+"(?<exp00>"+EXP+")"+
  b+";"+b+"\\k<ind0>"+b+"(?<relOp0>"+REL_OP+")"+b+
  "(?<exp10>"+EXP+")"+b+";"+ITER0_INCREASE+"\\))\\s*+"
  // body in brackets
  + "(?:\\{\\s*?" +
  "(?<blockFiOu>"+BLOCKR+"\\s++)" +
  "(?<lineAssiA>"+ACCUA_ASSIGNMENT+";)\\s*+" +
  "(?<blockSeOu>"+BLOCKP+")\\s*+" +
  "\\}"
  // no brackets
  + "|" +
  "(?<lineAssiB>"+ACCUB_ASSIGNMENT+";))"
);
```

We implemented the 10 patterns listed in Table 2 to detect SP-formula code. On average, the regular expressions for non-nested loops consist of 8 lines of code without comments, those for nested loops of 28 lines.

Table 2: Patterns implemented by the tool

Patterns for non-nested loops	FIS	`for`-loop for sum/product
	FES	`foreach`-loop for sum/product
	FIA	`for`-loop for arrays
	FEC	`foreach`-loop for arrays/collections
Patterns for nested loops	NFISS	`for`-loops for sum/product of sums/products
	NFESS	`foreach`-loops for sum/product of sums/products
	NFIAS	`for`-loops for array of products/sums
	NFECS	`foreach`-loops for array/collection of products/sums
	NFIAA	`for`-loops for array of arrays
	NFECC	`foreach`-loops for array/collection of arrays/collections

Evaluation. To evaluate our approach to detect SP-formula code in software repositories we applied our tool with the patterns listed in Table 2 to random samples of Java files on GitHub. As validation metrics we use recall and precision:

- Recall: How many of the SP-formula code fragments in a sample are automatically detected?
- Precision: To what extent are the detected SP-formula code fragments correct?

More precisely, let F be the set of all formula code fragments in the sample, and D be the detected formula code fragments. Then the recall is the number of correctly found formula code fragments divided by all correct formula code fragments, i.e., $\text{recall} = \frac{|D \cap F|}{|F|}$, and the precision is the number of correctly found formula code fragments divided by all found formula code fragments, i.e., $\text{precision} = \frac{|D \cap F|}{|D|}$. Based on the GBQ GitHub and GHTorrent dataset, we retrieved a list of 21,052,682 Java filenames (including full path information) on GitHub. The list does not include any forked repositories and duplicates, i.e. files with the same hash value. To draw random samples from the above data set we used the statistical programming language R and its uniform distributed function 'runif'. To measure the recall of our approach we used a sample of 1,000 Java files, to measure its precision we used a sample of 10,000 files. Since the GBQ GitHub and GHTorrent datasets are off-line mirrors of GitHub metadata, each sample contained names of files which have already been moved, removed or renamed on GitHub. Thus, we could not download the source code of all files leading to a sample size of 878 and 8,960, respectively.

Oracle. For computing the recall, we created an oracle data set by manually inspecting the complete sample of 878 files and annotating the formula code fragments that we found. We annotated the code fragments in a group discussion in order to reduce subjective biases. To annotate the fragments, we enclosed them in XML-tags which can be nested. The choice of the tags `<SimpleNestedLoop>`, `<DoubleNestedLoop>`, `<SimpleFormula>`, `<Assignment>`, `<Matrix>`, and `<Vector>` is based on the results of our keyword-based search. Finally, we manually found and annotated 145 formula code fragments in 53 of the 878 files (6.04%). These formula code fragments made up 1,064 lines of 142,419 total lines of code (0.75%). These annotated formula code fragments contained SP-formula code in 110 cases (75.86%). Almost all remaining cases were annotated with `<SimpleFormula>` which we used to describe simple, but non-trivial arithmetic expressions. The mathematical representation of those expressions may have some added value compared to the representation in program code, e.g. $\sqrt{5}$ instead of `Math.sqrt(5)`. Matrices and vectors were only used in 4 lines outside of loops.

Recall. Our tool found 34 of 110 SP-formula code fragments in the oracle data set. Thus, the recall is 30.91%. In cases where both an inner and outer loop each represent an SP-Formula code fragment, the tool would only consider the outer one. However, in our evaluation this case never happened.

Precision. To measure the precision we applied our tool to the 8,960 Java files of the second, non-annotated sample. For each detected formula code fragment, we manually checked whether the matched code fragment implemented an SP-formula. We also recorded whether the inferred formula covered the matched code fragment completely or only a part of it.

Our tool found 181 matches. All of these matches contained SP-formula code. 153 code fragments got completely specified by the inferred formula in mathematical notation. For 23 matches the tool inferred a correct formula that, however, did not describe the whole code excerpt. Only in 5 cases we found that the inferred mathematical formula was inadequate or wrong (2.76%). Thus, if we only take into account whether the match contained formula code, the precision was 100%. If we require that the inferred formula is correct, the precision was 97.23%, and if we require that the inferred formula is correct and completely describes the effect of the code matched, the precision was 84.53%.

Formula code density. To quantify how often formula code occurs in real world software (RQ2), we define two different measures for the formula code density—one based on lines of codes ρ_{LOC} and one based on number of unique files ρ_{files}:

$$\rho_{files} = \frac{\text{\#files containing formula code}}{\text{\#all scanned files}} \tag{32}$$

$$\rho_{LOC} = \frac{\text{\#lines with formula code in all scanned files}}{\text{\#lines of all scanned files}} \tag{33}$$

The formula code densities in our oracle data set, i.e. based on all manually identified and annotated formula code fragments, were $\rho_{files} = \frac{53}{878} = 6.04\%$ and $\rho_{LOC} = \frac{1{,}064}{142{,}419} = 0.76\%$. In other words, in our oracle data set on average one of 130 lines of code was part of a code fragment which we annotated as formula code.

5.2. SP-Formula Code in engineered Java software projects on GitHub

In the following, we introduce the examined sample of randomly chosen open source Java stargazer projects, present results of our detection tool for this sample and look at the application domains of the SP-formula code-rich projects.

Stargazers. We used the stargazers-based classifier approach with threshold 10 which according to a recent study by Munaiah et al. [36] has high precision (97%) and a reasonable recall (32%) to predict whether a GitHub project is an engineered software project and thus is sufficient for our purposes. First, we generated a sample of randomly chosen non-forked open source Java projects from GitHub excluding the already examined projects from our preliminary qualitative study. We utilized the GBQ GHTorrent dataset to retrieve the initial project list which contained 255,561 projects (as of January, 8th 2019). Next, we filtered the list by watcher (Stargazers) count greater than 10. The filtered list contained 28,139 projects, from which we randomly drew 1000 with the help of the statistical programming language R. The results of applying our SP-formula code detection tool to the Stargazers sample, the SciC (scientific computing) sample (see Subsection 5.3) as well as aggregated results are shown in Table 3. It lists the total number of projects investigated in each sample (#projects), the number of projects containing any kind of Java code at all (#nonempty), the number of projects which contained any SP-formula code detected by the tool (#fc projects), the total number of Java files within the complete sample (#files), the total number of files which contained any SP-formula code detected by the tool (#fc files), the total number of lines of Java code in the complete sample (LOC), the total number of detected lines of SP-formula code (LOFC), the total number of matches found by the tool in the complete sample (#matches) as well as code densities for actually detected SP-formula code ρ^{SP}_{files} and ρ^{SP}_{LOC} based on the definitions of general formula code densities in Equation 32 and 33, respectively. Assuming, that the distribution of SP-formula code in the samples is the same as in the oracle, we can use the recall of 31% determined in Section 5.1 to compute a rough estimation of the real SP-formula code densities $\widetilde{\rho}^{SP}_{LOC} = \frac{1}{\text{recall}}\,\rho^{SP}_{LOC}$, resp. $\widetilde{\rho}^{SP}_{files} = \frac{1}{\text{recall}}\,\rho^{SP}_{files}$.

LOC was computed with the Unix command cloc (version 1.74) and does neither include comments nor performs a uniqueness test on files. The same holds for LOFC computed by our tool, since it removes comments before scanning the files and processes every file in the project. Further statistical analyses were done with R. In total, we scanned 199,457 Java files from 949 open source Java projects of arbitrary topic available on GitHub using the patterns listed in Table 2. Below, we present the results for this sample.

Table 3: Results of the SP-formula code detection tool for each sample

Sample	Stargazers	SciC	Sum
#projects	1000	14	1014
#nonempty	949	14	963
#fc projects	266	11	277
#files	199,457	4050	203,507
#fc files	1713	199	1,912
LOC	30,275,938	548,976	30,824,914
LOFC	13,094	1,794	14,888
#matches	2,858	515	3,373
ρ^{SP}_{files}	0.85%	4,91%	0.94%
ρ^{SP}_{LOC}	0.043%	0.32%	0.048%
$\widetilde{\rho}^{SP}_{files}$	2.74%	15.84%	3,03%
$\widetilde{\rho}^{SP}_{LOC}$	0.14%	1,03%	0.15%

Results for Stargazers. Our detection tool yielded 2,858 SP-formula code matches in 266 of 949 projects (28%) respectively in 1713 of 199,457 Java files in this sample. The detected SP-formula code was spread over 13,094 lines of 30,275,938 total lines of Java code. Thus, the densities of SP-formula code in this sample are $\rho^{SP}_{files} = 0.85\%$ and $\rho^{SP}_{LOC} = 0.043\%$. Practically speaking, on average the tool detected SP-formula code in one of 117 files respectively in one of 2325 lines of code. As can be seen in Figure 3, every pattern derived from our preliminary study occurred in the Stargazers sample, which confirms their relevance. Nevertheless, the non-nested loop patterns FIS, FES and FIA are more common than the nested ones. Compared to the other non-nested loop patterns, the pattern FEA describing a foreach-loop that traverses an array sticks out, because it is rarely found.

Considering the absolute number of matches is not sufficient to tell if a project contains much SP-formula code. Therefore, we further investigate the SP-formula code density, ρ^{SP}_{LOC} introduced earlier in this section to identify SP-formula code rich projects. A corresponding scatter plot is presented in Figure 2. It becomes apparent that only few projects (18, with a density ρ^{SP}_{LOC} greater or equal to 0.01) have comparatively high formula code densities, i.e. at least 1 of 100 lines of code is part of SP-formula code. Table 4 shows the complete list of these projects.

The 18 projects with high SP-formula code density form the basis for a coding of the respective application domains and thus towards further investigations concerning RQ2.

Application Domains. As there was no sufficiently exact and efficient way to automatically determine the application domain of GitHub projects, we manually inspected the web sites of the projects to extract this information. While GitHub offers the feature *topics* [34], this feature is not sufficiently informative for the majority of projects examined in this work.

As it was not possible with reasonable effort to determine the application areas for all projects included in our sample, we decided to take a closer look at the application domains of the projects having a high SP-formula code density, more precisely, the top 18 projects of the Stargazers sample in terms of ρ^{SP}_{LOC} as presented in Table 4. Again we applied open and axial coding to determine the application domains based on the descriptions of these projects on GitHub and names of code artifact, such as classes and packages. In total, we performed two iterations involving three of the four authors, where we abstracted from more specific to more generic

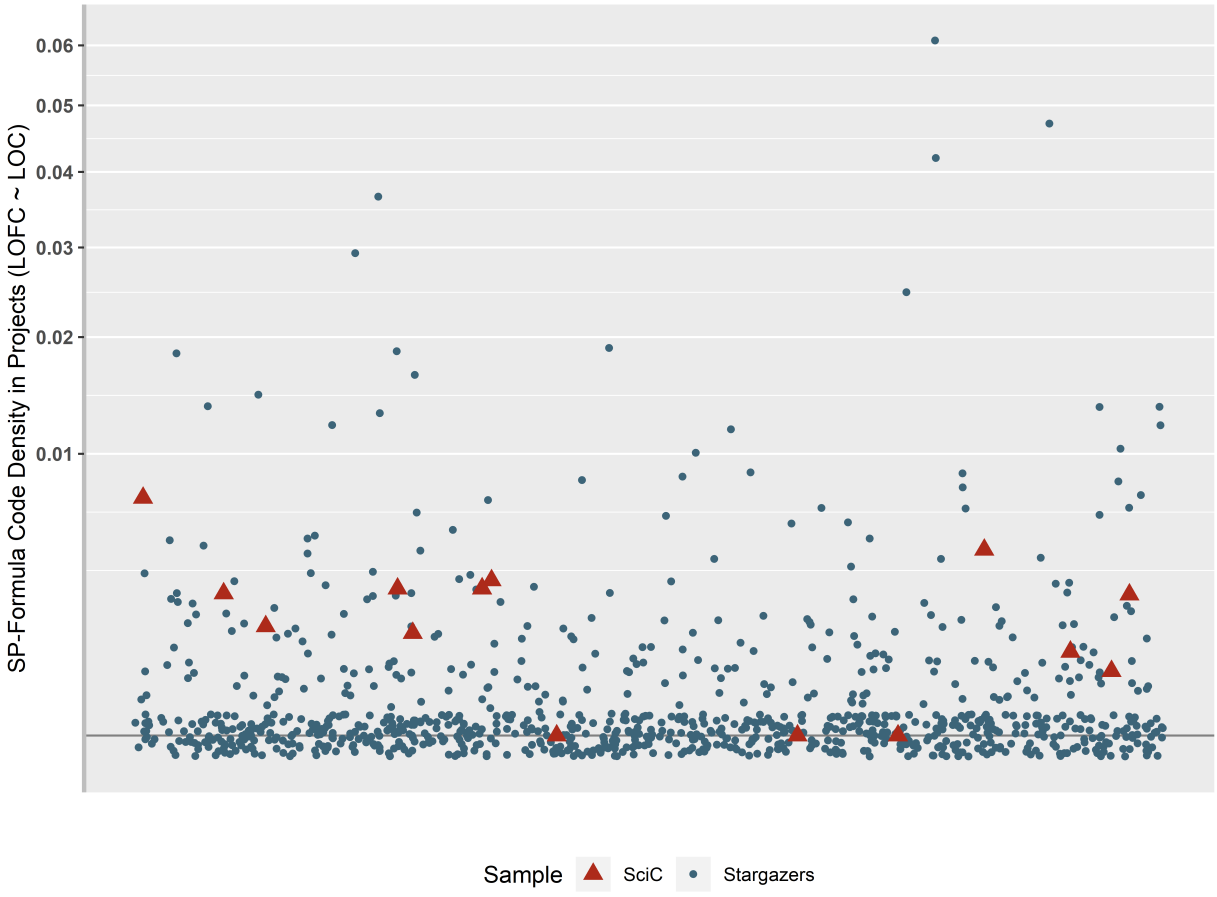

Figure 2: Jittered scatter plot of SP-formula code densities ρ_{LOC}^{SP} across all projects for each sample.

Table 4: Top 18 projects of the Stargazers sample having high (greater or equal to 0.01) SP-formula code densities ordered by ρ_{LOC}^{SP} including their respective coded application domain.

Project	LOFC	LOC	ρ_{LOC}^{SP}	Application Domain
grunka/Fortuna	87	1476	0.058	Cryptography
hfut-dmic/ContentExtractor	16	355	0,045	Information Retrieval
eljefe6a/UnoExample	3	75	0,040	Programming Model
radzio/AndroidOggStreamPlayer	272	7068	0,038	Signal Processing
brendano/myutil	319	10930	0,029	Statistics
InfiniteSearchSpace/Automata-Gen-3	100	3694	0,027	Simulation
DASAR/Minim-Android	186	9604	0,019	Signal Processing
hanks/Natural-Language-Processing	79	4358	0,018	Natural Language Processing
liuxang/LivePublisher	11	636	0,017	Signal Processing
sudohippie/throttle	12	730	0,016	Computer Networks
jlmd/SimpleNeuralNetwork	6	423	0,014	Machine Learning
ozelentok/CodingBat-Soultions	36	2593	0,013	Programming Language Practice
aliHafizji/CreditCardEditText	4	307	0,013	Mobile User Interface
quiqueqs-BabushkaText	4	318	0,012	Mobile User Interface
jroper-play-promise-presentation	3	239	0,012	Programming Model
jestan/netty-perf	8	669	0,011	Computer Networks
chipKIT32/chipKIT32-MAX	674	62409	0,010	Micro-Controller
martijnvdwoude/recycler-view-merge-adapter	3	281	0,010	Mobile User Interface

categories. The resulting application domains are also shown in Table 4.

The results of the application domain coding do partially coincide with our expectations. One SP-formula code dense project is a practice project, where features of the Java programming language are exercised. Another two projects highlight a certain programming model. Furthermore, three projects are concerned with mobile user interface elements. These projects form outliers in the sense that their high formula code density is due to their low number of lines of code, because they only contain a single SP-formula code match (see Table 4).

The other twelve projects draw a different application domain picture. We labeled three projects with Signal Processing as they dealt with either audio or video encoding, respectively transfer. Besides that, we assigned the labels Computer Networks, Statistics, Information Retrieval, Neural Networks, Natural Language Processing, Micro-Controller, Simulation and Cryptography to characterize the application domains of the remaining SP-formula code dense projects. For each of the labels assigned to these twelve projects, we can associate the application of mathematical formulas. Surprisingly, neither domains like computer graphics and games, nor chemistry and physics, which one typically subsumes by the term scientific-computing, occurred among the projects with the highest SP-formula-code density in our sample. Nonetheless, the majority of the assigned labels seem related to scientific-computing. Thus, projects of this application domain seem promising in terms of frequency of SP-formula code and we investigate a thematically more focused sample as described in the following.

5.3. SP-Formula Code in Scientific-Computing Java Projects on GitHub

Due to the unexpected distribution of application domains among the projects with the highest SP-formula-code density, we drew another sample solely consisting of Java repositories with topic *scientific-computing*.

SciC. To generate this topic-specific sample directly from the GitHub website, we used the search term `language:Java topic:scientific-computing` to query Java repositories. The 14 resulting repositories (as of January, 9th 2019) do neither intersect with the already examined projects of our preliminary qualitative study nor with the sample Stargazers.
We followed the exact same approach to compute the data for the sample SciC as we did for the other sample. The results are listed in the second column in Table 3.

Results for SciC. A total of 4050 Java files and 548,976 lines of code were scanned in this sample. The tool detected 515 SP-formula code matches in eleven of 14 projects (78.5%) in 199 different Java files and 1,794 lines of SP-formula code. Hence the density of detected SP-formula code based on files and lines of code is $\rho^{SP}_{files} = 4.91\%$ and $\rho^{SP}_{LOC} = 0.32\%$, respectively. In Figure 2 we can see that 11 of the 14 projects have a considerably higher SP-formula code density than most of the projects in the Stargazers sample. Furthermore, Figure 2 reveals that six of the implemented patterns also occur in this sample. Note, that in contrast to the Stargazers sample, here the two patterns NFIAA and NFIAS have a high density.

6. Discussion

6.1. Diversity of Formula Code

From our qualitative study, we can conclude that there exists a wide range of formula code. On that basis, we can give first answers to research question **RQ1**: One general observation is that the full extent of an implemented formula was often not directly recognizable. For some samples in our qualitative study, it took considerable effort to track down the complete implementation of a documented formula. Especially when the code fragments were split across different source code artifacts such as classes or files. We manually inspected 142 matches and classified 47 as real formula code. While the formula code was certainly diverse, almost half of the code involved `for`-loops, and also almost half of the code used arrays. We found several examples of incremental implementations of formulas. In these cases, the code may only implement the formula when we assume a certain dynamic behavior, such as a certain sequence of method calls or object instantiations. Detecting these kinds of formula implementations and asking the programmer to add assertions to the code to assure the correct dynamic behavior could help to prevent erroneous usage. Furthermore, reconstructing a formula in mathematical notation from the code was a very valuable and intuitive part in the process of code comprehension. From that point of view, it is obvious that one integral requirement for formula code support is the visual mathematical representation of the respective code.

Splitting. We also found that very often, complex expressions are split into multiple partial computations storing the interim results in temporary variables — either with names that intend to describe the computational part it represents, or with simple names such as `tmp`. The developers often followed an approach in which they split the expressions by operator precedence. For example, fractions the numerator and denominator computations are separated, stored in temporary variables and then divided in a subsequent statement. Besides the motivation of making the code more reusable and readable, developers seem to also split large expressions to facilitate debugging of the formula code. The splitting of computations makes it possible to leverage interactive break-point debuggers to investigate the interim results as well as assure correct application of arithmetic operations and functions according to their sequence and precedence. Current interactive debuggers only support line-by-line evaluation of code. A more fine grained approach in

which single operations in the same line can be investigated seems helpful. For formula code, an interactive mathematical visualization where parts of a formula could be collapsed, expanded, evaluated, used as break-points and also be edited with respective effectual code changes would form a promising extension to current debugging mechanisms.

Naming and Formatting. The scope of arithmetic computations is not limited to scalar values. We often found groups of variables coherent in the way they were named and the kind of operations applied to them. These variables were actually used to represent a single element of a vector or matrix. We found these groups of variables being modeled either as object variables encapsulated in a class and thus programmatically specifying their coherence, or completely unbounded as local variables. The formatting of corresponding code snippets gives the impression that developers try to visually form a vector or matrix in a known mathematical way. For vectors, we often discovered a vertical arrangement such that every component of a vector is computed subsequently in its own line of code. For matrices, the variables are sub-grouped for every row and column in a similar approach as for vectors, i.e. for each row- and column vector separately. This gives us even more evidence that developers want to have a visual representation of the formula code close to its mathematical representation.

Arrays. In total, 36 out of the 47 investigated code examples in our qualitative study were concerned with vector or matrix computations. Besides the low level modeling of vectors and matrices through semantic groups of variables, arrays are being used for this purpose. Surprisingly, also one-dimensional arrays are being used to model matrices. It is possible that developers want to avoid the computational overhead involved with n-dimensional arrays. In Java an n-dimensional array is actually a one dimensional array with pointers to (n-1)-dimensional arrays. Hence, we assume that developers intend to trade computing offsets for dereferencing pointers. Among others, this represents one phenomenon where we encountered a performance optimization at the expense of the formula's recognizability and thus overall code readability and maintainability.

Loops. Along with the utilization of arrays comes the application of loops, not only to iterate through arrays, but also, and in particular, to implement sum and product formulas. Those kinds of formulas occurred not only in vector/matrix contexts but also in scalar computations and sequences. In our small sample in Section 3, `for`-loops with an indexing variable were the predominant kind of loops used. This is not surprising, since these are already syntactically very close to the $\sum$ or $\prod$ operators in mathematics. Nested conditionals within the loops body could directly be translated to a part of the mathematical representation. Only in a few cases the formula code contained extra code, for example debug statements.

While the `while`-loop plays a secondary role in our findings of the qualitative study, the `foreach`-loop turns out to be relevant in the context of formula code as well. `foreach`-loops are mostly applied on collections and in particular to traverse them where the sequence of processing the elements does not play a significant role. In the formula code context, `foreach`-loops can not only be used to implement sum and product formulas over collections, but also to implement logical formulas (predicates) related to these collections using the universal quantifier $\forall$ and existential qualifier $\exists$. The derived code patterns for sum and product formulas of Section 4 concentrate on `for` and `foreach`-loops in simple and nested variants.

6.2. Frequency of Formula Code

To answers research question **RQ2**, we decided to investigate the frequency of formula code for sums and products in terms of formula code densities as defined in Section 5.1. These metrics are relative to the size of the projects and thus give a better impression than absolute numbers of matches and are comparable between projects. Above all, we found a 7.4 times higher SP-formula code density in sample of scientific-computing projects compared to the one of engineered software projects. Within the latter sample, we also found the SP-formula code-rich projects came from application domains related to scientific-computing. Nonetheless, the small size of the SciC sample calls for repeating the study as soon as more scientific-computing Java projects become available on GitHub.

Estimations. Based on our detection tool's recall, we determined a rough estimation of SP-formula code density of $\widetilde{\rho}^{SP}_{LOC} = 0.14\%$ for the Stargazers sample and $\widetilde{\rho}^{SP}_{LOC} = 1.03\%$ for the SciC sample. In other words, we estimate that about 1 of 700 lines of code in an engineered Java software project and 1 of 100 lines in a scientific-computing Java software project is part of an implementation of a sum or product formula. What does this mean in practice? The daily programming tasks of a software developer are not limited to writing code. Tasks related to the project's code base comprise writing, editing and, to a major proportion, reading and with that comprehending the code. Thus, we are certain that an average software developer gets in touch with SP-formula code multiple times a work week.

Patterns. In our quantitative study, we have shown the relevance of all patterns in the context of formula code. Figure 3 reveals that in the Stargazers sample, every SP-formula code pattern was detected at least once. In the SciC sample, six of the eight patterns appeared. Furthermore, the three SP-formula code patterns FES, FIA and FIS (all based on non-nested `for`-loops) are the most frequent patterns in both samples (Figure 3). We determine a 3.7, 46.2 respectively 10.6 times higher density of these patterns in the SciC sample. Besides that, we discover higher densities for the nested SP-formula code patterns in the SciC sample as well. This represents another insight showing an increased relevance of the patterns as well as an increased probability to encounter SP-formula code in scientific-computing projects.

Percentage of loops implementing formulas. To put the SP-formula code density into perspective, we investigated how many (nested) `for`-loops occur in the source code and how many of these actually implement SP-formulas. To this end, we implemented two patterns using regular expressions, similar to the SP-formula code patterns, in order to detect simple (non-nested) and nested `for`- and `foreach`-loops. In the following, we use the term loops to refer to both `for`- and `foreach`-loops. We applied our detection tool (Subsection 5.1) with the new patterns to ensure comparability. Table 5 summarizes the detection tool's results for the general loop patterns. We report for both simple and nested loops the total number of matches in the whole sample (#matches), the total number of files in which a match occurred (#files) and the total number of projects in which a match occurred (#projects). The tool yielded 114.793 simple and 9,558 nested loops in the Stargazers sample. Surprisingly, in 156, respectively 534 of the 946 projects, i.e. in 16,49%, respectively 56.44%, no simple, respectively nested loops were found. In contrast, 2,738 SP-formula code matches, based on simple loops and 120 SP-formula code matches based on nested loops were detected in this sample. Therefore, 2.38% (every 42nd) of all simple loops and 1.25% (every 80th) of all nested loops implement a sum of product formula according to our definition. When we apply a rough estimation depending on the detection tool's recall, in 7.71%

Table 5: Results of the detection tool for each sample utilizing the general loop patterns.

	Simple `for`-loops			Nested `for`-loops		
Sample	#matches	#files	#projects	#matches	#files	#projects
Stargazers	114,793	38,143	790	9,558	5,278	412
SciC	6,350	1,255	13	1,685	460	10

(every 13th) a simple and in 4.06% (every 25th) a nested loop implements a SP-formula. In the SciC sample, we found 6,350 simple and 1,685 nested loops. The SP-formula code matches in this sample amount to 483 simple and 32 nested loops. Thus, 7.60% (every 13th), respectively 1.89% (every 52nd) of all investigated simple, respectively nested loops form a SP-formula according to our definition. Taking the detection tool's recall into account, we roughly estimate that 24.60% (every 4th) for simple, and 6.14% (every 16th) for nested loops implement sum or product formulas.

Overall, we think that it is reasonable to assume that formula code in scientific-computing is more frequent, more diverse and more complex than in most other application areas.

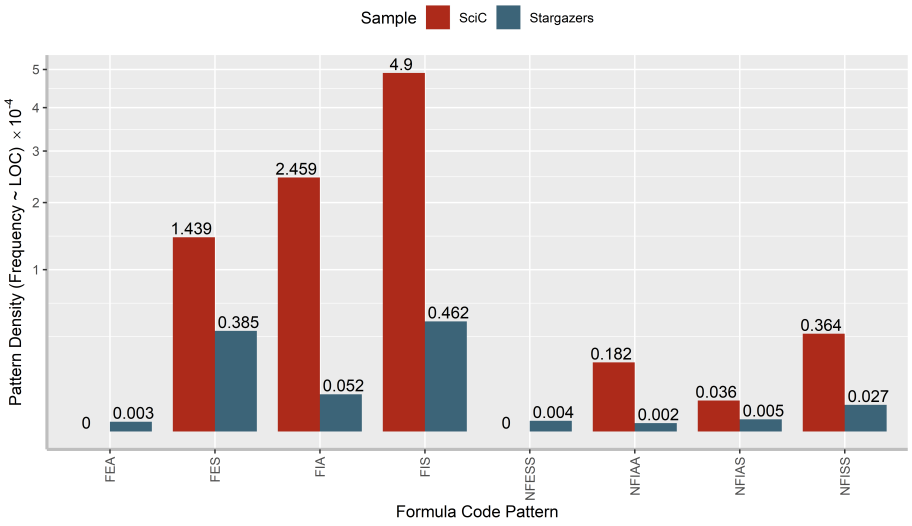

Figure 3: Density of SP-formula code matches for each pattern relative to LOC per sample.

7. Limitations

To classify a code fragment as formula code, we do not require that the programmer's original intention was to implement a mathematical formula, but instead it is sufficient that the implemented computation could be expressed by a mathematical formula. On the other hand, some program code fragments that really implement a mathematical formula might not directly be expressible in a mathematical notation since the respective code is already too far away from the maths, possibly due to performance optimizations, refactorings or applied coding 'hacks'.

Each of the empirical studies presented in this paper comes with its own limitations.

Qualitative Analysis of Formula Code Examples. The formula code examples that we manually inspected were selected based on keywords occurring in their comments. Undocumented formula code, i.e. code that had none of our keywords in its comments, may have different properties. Furthermore, while our selection of keywords certainly introduces a bias toward sum and product formulas, it did not affect the oracle data set, since we annotated any formula code that we found. We also tried to increase credibility and transferability by following established coding methods and by discussing all reconstructed formulas in group sessions with all authors. This also holds for our coding of the application domains.

Quantitative Tool Evaluation. While the tags that we used to annotate the oracle data set were based on the results of the qualitative study, we did not exclude any kind of formula code only because it was not considered in the qualitative study. However, the annotation of the oracle data set depends on the subjective assessment whether some code fragment implements a formula.

Quantitative Analysis of SP-formula code on GitHub. We tried to carefully distinguish between the SP-formula code detected by our tool and the SP-formula code actually present in a sample. We used a randomized, stargazer-based sampling strategy to increase the generalizability of our findings to engineered software projects on GitHub. The generalizability of our results for the sample SciC is strongly limited by the small size of the sample.

To enable other researchers to verify our results, we provide all data that is not protected by copyright (e.g. content of GitHub repos) as supplementary material [37].

8. Conclusion

So far, research in software engineering has focused on the synthesis but not the analysis of formula code. In this paper, we first presented a qualitative study designed for gaining insights into the diversity of formula code in real world Java projects (**RQ1**). We found that code that developers document as formula code has special properties. The observed phenomena range from coded mathematical notation in the names of variables, complex arithmetic operations split by a debugging or precedence strategy, groups of variables with coherent naming and coherently applied operations, over sum and product formulas based on `for`-loops up to incremental formula implementations that depend on the dynamic behavior (call sequence). In particular, we find it promising to provide an alternative representation of the respective code in terms of a mathematical notation. During our qualitative study, deriving such a mathematical representation of the code supported the process of code comprehension to a great extent. If those code visualizations are made interactive and reachable directly from within the source code editor, we assume that they would greatly facilitate debugging of formula code. Designing and implementing such debugging features and assessing their usability and efficiency are part of our plans for future research.

Furthermore, we presented an approach to detect SP-formula code using syntactic patterns in combination with a set of constraints on the variables occurring in the matched code fragments. We derived these patterns based on our preliminary qualitative study and evaluated the effectiveness of our approach in terms of recall and precision. On that basis, we performed a case study to investigate the frequency of SP-formula code in a sample of 1,000 open source Java stargazer projects on GitHub (**RQ2**). We also looked at the application domains of the SP-formula code-rich projects and found a major overlap with scientific, respectively technical

subject areas, e.g. information retrieval, signal processing, computer networks, statistics, machine learning and simulation. This inspired us to also apply our tool to a sample consisting solely of scientific-computing projects (SciC). Since they give a more realistic impression on the real distribution of SP-formula code in open source Java projects, here, we only give densities estimated based on our detection approach's recall. The absolute numbers have been presented and discussed in the previous sections. We estimate that one of 700 lines of code in the Stargazers and even one of 100 lines of code in the SciC sample is part of an implementation of a sum or product formula. In addition to the line-based metrics, we also computed the total number of simple and nested `for`-loops and `foreach`-loops in the samples. We estimate that in the stargazers sample every 13th simple respectively 25th nested loop implements a sum or product. In the SciC sample the ratio is considerably higher: every 4th simple respectively 16th nested loop. Based on these numbers, it is reasonable to assume that an average software developer will have to write, or at least comprehend, SP-formula code multiple times a work week.

As we determine a 7.4 times higher SP-formula code density in the SciC sample, we think that the design of tools and language features specialized for formula code in this domain is a promising route for future research. Thus, we intend to enhance and extend the patterns in our tool, e.g., by adding vector and matrix patterns, and in particular, we want to investigate other programming languages such as Python, which are more common in the field of scientific-computing.

References

[1] F. Cajori, A history of mathematical notations, The Open Court Publishing Co., Chicago, IL, 1929.

[2] Matt Lake, Epic failures: 11 infamous software bugs — computerworld, https://www.computerworld.com/article/2515483/enterprise-applications/epic-failures--11-infamous-software-bugs.html?page=2, Accessed October 2, 2018, 2010.

[3] Jonathan P. Leech, Larry Klaes, Space faq 08/13 - planetary probe history, http://www.faqs.org/faqs/space/probe/, Accessed October 2, 2018, 2017.

[4] Marty Moore, The risks digest volume 5 issue 73, http://catless.ncl.ac.uk/Risks/5.73.html#subj2, Accessed October 2, 2018, 1987.

[5] P. B. Henderson, Mathematical reasoning in software engineering education, Commun. ACM 46 (2003) 45–50. URL: https://doi.org/10.1145/903893.903919. doi:10.1145/903893.903919.

[6] D. Landy, C. Allen, C. Zednik, A perceptual account of symbolic reasoning, Frontiers in Psychology 5 (2014) 275. URL: http://journal.frontiersin.org/article/10.3389/fpsyg.2014.00275. doi:10.3389/fpsyg.2014.00275.

[7] B. W. Kernighan, L. L. Cherry, A system for typesetting mathematics, Commun. ACM 18 (1975) 151–157. URL: http://doi.acm.org/10.1145/360680.360684. doi:10.1145/360680.360684.

[8] D. E. Knuth, Mathematical typography, Bull. Amer. Math. Soc. 1 (1979) 337–372. URL: http://projecteuclid.org/euclid.bams/1183544082. doi:10.1090/S0273-0979-1979-14598-1.

[9] M. Levison, Editing mathematical formulae, Softw., Pract. Exper. 13 (1983) 189–195. URL: https://doi.org/10.1002/spe.4380130208. doi:10.1002/spe.4380130208.

[10] R. Zanibbi, D. Blostein, Recognition and retrieval of mathematical expressions, International Journal on Document Analysis and Recognition (IJDAR) 15 (2012) 331–357. URL: http://dx.doi.org/10.1007/s10032-011-0174-4. doi:10.1007/s10032-011-0174-4.

[11] K.-F. Chan, D.-Y. Yeung, Mathematical expression recognition: a survey, International Journal on Document Analysis and Recognition 3 (2000) 3–15. URL: http://dx.doi.org/10.1007/PL00013549. doi:10.1007/PL00013549.

[12] Jetbrains, Mps: Domain-specific language creator by jetbrains, https://www.jetbrains.com/mps/, Accessed May 25, 2018, 2018.

[13] S. Breu, T. Zimmermann, C. Lindig, Mining eclipse for cross-cutting concerns, in: Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006, 2006, pp. 94–97. URL: https://doi.org/10.1145/1137983.1138006. doi:10.1145/1137983.1138006.

[14] T. Xie, J. Pei, MAPO: mining API usages from open source repositories, in: Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006, 2006, pp. 54–57. URL: https://doi.org/10.1145/1137983.1137997. doi:10.1145/1137983.1137997.

[15] T. Zimmermann, P. Weißgerber, S. Diehl, A. Zeller, Mining version histories to guide software changes, IEEE Trans. Software Eng. 31 (2005) 429–445. URL: https://doi.org/10.1109/TSE.2005.72. doi:10.1109/TSE.2005.72.

[16] P. Weißgerber, S. Diehl, Identifying refactorings from source-code changes, 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan, 2006, pp. 231–240. URL: https://doi.org/10.1109/ASE.2006.41. doi:10.1109/ASE.2006.41.

[17] M. Moser, J. Pichler, Documentation generation from annotated source code of scientific software: position paper, Proceedings of the International Workshop on Software Engineering for Science, SE4Science@ICSE 2016, Austin, Texas, USA, May 14-22, 2016, 2016, pp. 12–15. URL: https://doi.org/10.1145/2897676.2897679. doi:10.1145/2897676.2897679.

[18] M. Moser, J. Pichler, G. Fleck, M. Witlatschil, Rbg: A documentation generator for scientific and engineering software, 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015, 2015, pp. 464–468. URL: https://doi.org/10.1109/SANER.2015.7081857. doi:10.1109/SANER.2015.7081857.

[19] S. Kamali, F. W. Tompa, Retrieving documents with mathematical content, The 36th International ACM SIGIR conference on research and development in Information Retrieval, SIGIR '13, Dublin, Ireland - July 28 - August 01, 2013, 2013, pp. 353–362. URL: https://doi.org/10.1145/2484028.2484083. doi:10.1145/2484028.2484083.

[20] R. Jain, S. Prathik, V. Vinayakarao, R. Purandare, A search system for mathematical expressions on software binaries, in: Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018, 2018, pp. 487–491. URL: https://doi.org/10.1145/3196398.3196413. doi:10.1145/3196398.3196413.

[21] A. D. Franco, H. Guo, C. Rubio-González, A comprehensive study of real-world numerical bug characteristics, in: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, 2017, pp. 509–519. URL: https://doi.org/10.1109/ASE.2017.8115662. doi:10.1109/ASE.2017.8115662.

[22] GitHub Inc., The state of the octoverse, https://octoverse.github.com/, Accessed December 18, 2018, ????

[23] IEEE, The 2018 top programming languages - ieee spectrum, https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018, Accessed October 11, 2018, 2018.

[24] TIOBE Software BV, Tiobe index — tiobe - the software quality company, https://www.tiobe.com/tiobe-index/, Accessed October 11, 2018, 2018.

[25] R. Dyer, H. A. Nguyen, H. Rajan, T. N. Nguyen, Boa: a language and infrastructure for analyzing ultra-large-scale software repositories, 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, 2013, pp. 422–431. URL: https://doi.org/10.1109/ICSE.2013.6606588. doi:10.1109/ICSE.2013.6606588.

[26] Google LLC, Bigquery - analytics data warehouse — google cloud platform, https://cloud.google.com/bigquery/, Accessed January 30, 2018, 2018.

[27] B. G. Glaser, A. L. Strauss, The discovery of Grounded Theroy: Strategies for Qualitative Research, Aldine Transaction, 1967.

[28] libgdx, Desktop/Android/HTML5/iOS Java game development framework, https://github.com/libgdx/libgdx/blob/master/gdx/src/com/badlogic/gdx/math/FloatCounter.java, Accessed May 27, 2018, 2018.

[29] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi-Reghizzi, D. Poshyvanyk, C. Fu, Q. Xie, C. Ghezzi, An empirical investigation into a large-scale java open source code repository, in: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, ESEM 2010, 16-17 September 2010, Bolzano/Bozen, Italy, 2010. URL: https://doi.org/10.1145/1852786.1852801. doi:10.1145/1852786.1852801.

[30] Airbnb, Inc, A machine learning package built for humans, https://github.com/airbnb/aerosolve/blob/master/core/src/main/java/com/airbnb/aerosolve/core/images/HOGFeature.java, Accessed May 27, 2018, 2018.

[31] J. Sajaniemi, An empirical analysis of roles of variables in novice-level procedural programs, 2002 IEEE CS International Symposium on Human-Centric Computing Languages and Environments (HCC 2002), 3-6 September 2002, Arlington, VA, USA, 2002, pp. 37–39. URL: https://doi.org/10.1109/HCC.2002.1046340. doi:10.1109/HCC.2002.1046340.

[32] A. Taherkhani, A. Korhonen, L. Malmi, Recognizing algorithms using language constructs, software metrics and roles of variables: An experiment with sorting algorithms, Comput. J. 54 (2011) 1049–1066. URL: https:

//doi.org/10.1093/comjnl/bxq049. doi:10.1093/comjnl/bxq049.
[33] A. Rountev, Precise identification of side-effect-free methods in java, 20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA, 2004, pp. 82–91. URL: https://doi.org/10.1109/ICSM.2004.1357793. doi:10.1109/ICSM.2004.1357793.
[34] GitHub Help, About topics - user documentation, https://help.github.com/articles/about-topics/, Accessed January 30, 2018, 2017.
[35] O. Moseler, F. Lemmer, S. Baltes, S. Diehl, On the Diversity and Frequency of Formula Code in Java: Supplementary Material (2019). URL: https://doi.org/10.5281/zenodo.1252323. doi:10.5281/zenodo.1252323.
[36] N. Munaiah, S. Kroh, C. Cabrey, M. Nagappan, Curating github for engineered software projects, Empirical Software Engineering 22 (2017) 3219–3253. URL: https://doi.org/10.1007/s10664-017-9512-6. doi:10.1007/s10664-017-9512-6.
[37] O. Moseler, F. Lemmer, S. Baltes, S. Diehl, On the Diversity and Frequency of Code Related to Mathematical Formulas in Real-World Java Projects, 2019. URL: https://doi.org/10.5281/zenodo.3566933. doi:10.5281/zenodo.3566933.