Discovering Math APIs by Mining Unit Tests

Anirudh Santhiar, Omesh Pandita[⋆], and Aditya Kanade

Department of Computer Science and Automation, Indian Institute of Science
{anirudh_s,pandita.omesh,kanade}@csa.iisc.ernet.in

Abstract. In today's API-rich world, programmer productivity depends heavily on the programmer's ability to discover the required APIs. In this paper, we present a technique and tool, called MathFinder, to discover APIs for mathematical computations by mining unit tests of API methods. Given a math expression, MathFinder synthesizes pseudo-code to compute the expression by mapping its subexpressions to API method calls. For each subexpression, MathFinder searches for a method such that there is a mapping between method inputs and variables of the subexpression. The subexpression, when evaluated on the test inputs of the method under this mapping, should produce results that match the method output on a large number of tests. We implemented MathFinder as an Eclipse plugin for discovery of third-party Java APIs and performed a user study to evaluate its effectiveness. In the study, the use of MathFinder resulted in a 2x improvement in programmer productivity. In 96% of the subexpressions queried for in the study, MathFinder retrieved the desired API methods as the top-most result. The top-most pseudo-code snippet to implement the entire expression was correct in 93% of the cases. Since the number of methods and unit tests to mine could be large in practice, we also implement MathFinder in a MapReduce framework and evaluate its scalability and response time.

1 Introduction

In today's API-rich world, programmer productivity depends heavily on the programmer's ability to discover the required APIs and to learn to use them quickly and correctly. Significant research efforts are therefore targeted at aiding programmers in API discovery. A programmer can search for APIs using a wide spectrum of techniques. They range from keywords [15, 3], types [16, 24], tests [11, 13], and code snippets [17], to formal specifications [29] or combinations of the above [21]. These approaches try to address the problem of API discovery in a general programming context and may face challenges in terms of precision of results or require programmers to invest too much effort in formulating the query (e.g., require a first-order logic specification).

In this paper, we address the problem of API discovery for mathematical computations. Mathematical computations are at the heart of numerous application domains such as statistics, machine learning, image processing, engineering or

[⋆] now at Microsoft India

scientific computations, and financial applications. Compared to general programming tasks, mathematical computations can be specified more easily and rigorously, using mathematical notation with well-defined semantics. Many interpreted languages like Matlab, Octave, R, and Scilab, are available for prototyping mathematical computations. It is a common practice to include prototype code to formalize math algorithms (e.g., in [25, 4]). The language interpreter gives a precise *executable semantics* to mathematical computations. Unfortunately, interpreted languages are not always suitable for integration into larger software systems of which the mathematical computations are a component. This is because of commercial and technical issues involving performance overheads, portability, and maintainability. In such cases, the programmer implements the mathematical computations in a general-purpose language.

General-purpose programming languages usually support only basic math operations. For example, `Java.lang.Math` supports elementary functions for exponentiation, logarithm, square root, and trigonometry. Advanced math domains are supported by third-party libraries. Availability of a number of competing libraries, their API sizes, and varying support for primitive operations make it difficult for programmers to select appropriate libraries. We present an approach for discovering math APIs to compute a given (set of) math expression(s). A programmer can pose expressions from the algorithm she wants to implement as queries. For example, suppose she asks for API methods to compute $\mathsf{v} = \mathsf{v}\ ./\ \mathsf{normf}(\mathsf{v})$ (where v is a matrix of doubles). Here, normf stands for the Frobenius norm, and ./ is matrix-scalar division.

Our technique, called MathFinder, returns pseudo-code to compute the expression by mapping subexpressions to method calls of individual libraries. For example, MathFinder identifies a method `double DoubleMatrix.norm2()` from a third party library as suitable for computing the subexpression $\mathsf{normf}(\mathsf{v})$. It identifies that v should be mapped to `this` and the result is available in the return value of the method call. MathFinder uses this mapping between variables in the subexpression and method parameters to emit an appropriate method call. In the synthesized pseudo-code, it declares an object v of type `DoubleMatrix`. This object corresponds to the variable v used in the expression. MathFinder emits `double T1 = v.norm2()` in the pseudo-code to implement $\mathsf{normf}(\mathsf{v})$. Here `T1` is a temporary variable. MathFinder also discovers if a method is likely to modify the input parameters (i.e. it discovers likely side-effects of methods). In this example, `norm2()` does not modify the receiver.

Discovering APIs and the information about setting up of parameters and determining side-effects automatically is a challenging problem. Formal specifications of semantics of methods may help us solve this problem. Specification languages like JML [12] are designed for annotating Java code with specifications. However, their use is not widespread yet. On the other hand, it is easy to get an under-approximate *operational specification* of a method in the form of *unit tests*. Unit testing is well-adopted in practice, supported by tools like JUnit[1]. We therefore use (the set of input/output objects in) unit tests as a de-

[1] junit.org

scription of method semantics. While we chose Java as the target programming language, our technique can work with other languages.

The key insight in MathFinder is to use an interpreter for a math language (such as Scilab[2]) to evaluate subexpressions on unit tests of library methods. The result of the interpretation on inputs of a test can be compared to the output of the test. Our *hypothesis* is that if a subexpression results in the same value as the output of a method on a large number of tests, the method can be used for computing the subexpression. The subexpression cannot directly be evaluated on data from unit tests because the math types used in the expression are independent from the datatypes used in library APIs. We therefore require the library developer to provide code to convert library datatypes to types of the math interpreter. Thus, any library developer can hook her library into MathFinder. In our running example, the library developer provides code to map objects of the type `DoubleMatrix` to double matrices used by the math interpreter. Writing code to convert library objects to data values of the interpreter is a one time task and was fairly straight-forward in our case.

Given an expression, MathFinder extracts subexpressions from it. Given a subexpression and a method, MathFinder computes the set of all candidate mappings between variables of the subexpression and method parameters, called *actuals-to-formals mappings*. The mappings should respect the correspondence between library datatypes and math types provided by the library developer. MathFinder then searches for a mapping that maximizes the number of unit tests on which the subexpression gives results equivalent to the method outputs. For example, there is only one possible actuals-to-formals map, (v, `this`), between the subexpression normf(v) and the method `norm2()`. We use the library developer's code to assign values contained in `this` to v. Then, normf(v) evaluates to a value equal to the return value of `norm2()` on every test of `norm2()` in a test-suite with 10 tests. Alongside, MathFinder also infers likely *side-effects* of a method call by comparing the input-output values of method parameters.

Our approach falls in the category of specification-driven API discovery [29, 21]. Unlike logical specifications used as queries in these approaches, the queries to MathFinder are executable and succinct. On the library developer's front, the specifications are easy to obtain – just the unit tests and a programmatic mapping from library datatypes to the math types. In contrast, in test-driven API discovery approaches [20, 8, 11, 21, 13], the programmer query is itself in the form of unit tests specific to a library. The unit tests are evaluated on library methods. Thus, the programmer has to know about library datatypes and invest time in writing unit tests. In our approach, the programmer query is independent of library datatypes (it uses mathematical types of the interpreted language). The same query applies to all libraries that are hooked into MathFinder. Other approaches cited above target API discovery for general programming tasks, whereas, we present a more specific approach for mathematical computations.

We have implemented MathFinder as an Eclipse plugin for discovering third-party Java APIs. We performed a user study to evaluate whether Math-

[2] scilab.org

Finder improves programmer productivity, when compared to standard practices such as the use of Javadoc, Eclipse code completion, and keyword-driven web or code search. All participants were permitted to use any of these techniques. On the same programming tasks, the participants who used MathFinder were twice as fast on average as those who did not use MathFinder.

MathFinders's results were quite precise across multiple libraries. The API method retrieved as the top-most result against a subexpression query was correct 96% of the time. The top-most pseudo-code snippet to implement the entire expression was correct in 93% of the cases. During the course of evaluating MathFinder, we found discrepancies between MathFinder's output and the Javadoc of JBlas library. While MathFinder indicated no side-effect on some methods, their Javadoc explicitly states that they perform computations "in-place"[3]. We studied the method implementations and found that the documentation was indeed inaccurate and the methods had no side-effects.

Our technique is inherently data-parallel. Since the test suite collection can be quite large in practice, we also implemented it in the `Hadoop`[4] MapReduce framework. It scaled to a large collection of unit tests consisting of over 200K tests and returned results in average 80.5s on an 8-core machine. These results are cached for real-time retrieval using the plugin.

We present an overview of MathFinder in the next section. We discuss the technique in Sections 3–4 and evaluate it in Section 5. We survey related approaches in Section 6. We sketch future directions and conclude in Section 7.

2 Overview

```
% input: matrix W of doubles, and
% double scalars d, v_error
pagerank(W, d, v_error)
  N = size(W, 2);
  v = rand(N, 1);
  v = v./normf(v);
  last_v = ones(N, 1)*INF;
  M_hat = d*W + (1−d)/N*ones(N, N);
  cur = normf(v−last_v);
  while( cur > v_error)
          last_v = v;
          v = M_hat*v;
          v = v./normf(v);
          cur = normf(v−last_v);
  end
```

Fig. 1: Scilab code for PageRank (adapted from Wikipedia)

In this section, we illustrate the MathFinder technique with an example. Consider the Scilab code in Fig. 1 for the PageRank algorithm [18], a ranking algorithm used by Google.

Even this reasonably small algorithm requires 9 matrix operators that are not supported by the standard Java library. The exact meaning of these operators is not critical for the present discussion. Selecting a third-party library that supports all of them is a tedious and time-consuming task. The four open-source Java libraries that we surveyed, namely, Colt, EJML, Jama, and JBlas[5] contain over 400 methods in

[3] jblas.org/javadoc/index.html, e.g., the `add` method
[4] hadoop.apache.org
[5] respectively, acs.lbl.gov/software/colt/, code.google.com/p/efficient-java-matrix-library/, math.nist.gov/javanumerics/jama/, jblas.org

the classes implementing double matrices. Of these, only JBlas (containing over 250 methods for matrix operations) supports all the required operators. The programmer must identify that JBlas is the right library. Further, the programmer must select appropriate methods and learn how to set up method parameters, and about side-effects of method calls, if any.

The programmer can use MathFinder to query for APIs to implement each expression in the algorithm. MathFinder gives an aggregate score to the libraries indicating how many of the required subexpressions can be implemented using methods from each library. The programmer can then easily identify JBlas as the only *functionally complete* library to implement PageRank.

Suppose the programmer wants to find out how to implement the assignment in line 6, v = v ./ normf(v) (discussed earlier in Section 1). In this paper, we use the math types *double* standing for double scalars, and *double M*, for double matrices. The variable v is given the type *double M* in the query

double M v; v = v ./ normf(v);

The expression form "LHS = RHS" indicates that the programmer wants methods to implement the RHS, and the types of the result and the LHS should be the same. Though many interpreted math languages perform dynamic type inferencing, we need type declarations to make the query unambiguous because operators used in these languages can be polymorphic. For example, ./ denotes both matrix-scalar division and element-wise division of matrices. For each library, its developer provides a mapping between math types and classes used in her library (like in Table 1), and code for converting values from the library's objects to values of the math type. This helps us translate type signatures and data between library types and math types. Thus, the queries themselves are independent of the target libraries.

Table 1: Library classes for the math type *double M*

Library	Class
Colt	`DenseDoubleMatrix2D`
EJML	`DenseMatrix64F`
Jama	`DoubleMatrix`
JBlas	`Matrix`

MathFinder then parses the math expression and decomposes it into subexpressions (similar to three-address code generation in compilers [1]). The subexpressions have a single operator on the RHS by default. v = v ./ normf(v) is decomposed as

double T1; *double M* v; T1 = normf(v); v = v ./ T1;

Operator precedence enforces the sequential ordering of computation and temporary variables like T1 are used to explicate data flow. Since the types of the operators are fixed by the chosen interpreted language, the types of the temporaries can be inferred. Here, MathFinder infers that T1 is a *double*. Our technique also permits the programmer to guide the search at a granularity other than individual operators in order to find a single API method to implement a larger subexpression, or even the entire expression.

Table 2: Results obtained against the subexpression T1 = normf(v)

Method	Actuals-to-formals Map	Score
`double Algebra.normF(DoubleMatrix2D)`	(v, `arg1`), (T1, return)	1.0
`static double NormOps.normF(D1Matrix64F)`	(v, `arg1`), (T1, return)	1.0
`static double NormOps.fastNormF(D1Matrix64F)`	(v, `arg1`), (T1, return)	1.0
`double Matrix.normF()`	(v, `this`), (T1, return)	1.0
`double DoubleMatrix.norm2()`	(v, `this`), (T1, return)	1.0
`static double NormOps.fastElementP(D1Matrix64F,double)`	(v, `arg1`), (T1, return)	0.3

MathFinder now picks each subexpression and mines unit tests of library methods to find the methods to implement it, along with the map from subexpression variables to the formal parameters of the method. The method parameters must range over library datatypes (or their supertypes) corresponding to the math types of the subexpression variables. The results obtained against the subexpression T1 = normf(v) are shown in Table 2. In the actuals-to-formals maps in Table 2, `arg1` stands for the first argument. MathFinder can also search for methods inherited from a superclass of a class identified by the library developer as implementing a math type. In this example, methods over `DoubleMatrix2D` and `D1Matrix64F` of the Colt and EJML libraries are also discovered. These are, respectively, supertypes of `DenseDoubleMatrix2D` and `DenseMatrix64F` identified in Table 1.

Recall our hypothesis that the relevance of a method to implement a subexpression is proportional to how often its unit tests match the subexpression on interpretation. We assign scores to methods based on this observation and rank them in decreasing order of their scores. As an example of a low-ranked method, we show the `NormOps.fastElementP` method of EJML in Table 2. It computes the p-norm and coincides with normf only on those tests that initialize its second parameter to 2. Its score (0.3) is much lower than the score of the methods that compute normf exclusively.

MathFinder then uses the results mined against each subexpression to issue a pseudo-code snippet. The snippet takes a set of input objects, returns an output object, and performs the computation queried for. The input objects correspond to variables from the RHS of the query and the output object, to the LHS variable. By convention, objects are given the same name as the variables they correspond to. A sequence of API method calls, with a call corresponding to every operator used in the query, is used to generate the output object. In this sequence, if the return value of a method is passed as an argument to another, then their library types should be compatible, and the output object is the return value of the last method. MathFinder suggests this snippet from JBlas

```
DoubleMatrix v; double T1; T1 = v.norm2(); v = v.div(T1);
```

where `div` is discovered to implement v = v ./ T1.

MathFinder can thus automate the process of API discovery and comprehension to a large extent. The programmer will still have to verify the validity of the results and translate the pseudo-code to Java code by introducing object instantiations as necessary. The programmer can use snippets from different libraries for different (sub)expressions in her algorithm, provided she writes code to convert between datatypes of the different libraries.

3 Problem Statement

In this section, we define the problem of math API discovery formally. Consider a query Q which is decomposed into sub-queries. A sub-query has type-declarations of variables, followed by a subexpression $x = e$, such that there is exactly one operator in e. We denote a sub-query by q. Given q, our objective is to find methods that can be used to implement e. Let m be a method such that there is a non-empty set $\Lambda(q, m)$ of actuals-to-formals maps, based on the type mapping given by the library developer.

Let $\lambda \in \Lambda(q, m)$ be an actuals-to-formals map . It maps variables in e to input parameters of m and maps the variable x to an output variable of m (either the return value or a parameter modified by side-effect). Let a unit test σ of m map m's input/output variables to Java objects. Let f be a function from Java objects to data values of the interpreter. The library developer programatically encodes f. Given a unit test σ of a method m and an actuals-to-formals map λ, for a subexpression $x = e$, σ' gives the values of variables occurring in $x = e$. For a variable y,

$$\sigma'(y) = f(\sigma(y')), \text{ where } y' = \lambda(y).$$

The mapping σ' can be extended to expressions in a natural way. For example, $\sigma'(\mathsf{normf}(\mathsf{v})) = \mathsf{normf}(\sigma'(\mathsf{v}))$, where the interpreter computes normf. The subexpression $x = e$ *evaluates* to true on a unit test σ, under an actuals-to-formals map λ, if $\sigma'(x) = \sigma'(e)$. A sub-query evaluates to true on a unit test if its subexpression does. Let N be the total number of unit tests of m, and k the number of tests on which q evaluates to true, under a particular actuals-to-formals map λ. The problem is then to find an actuals-to-formals map λ^* that maximizes k/N. We call λ^* the *maximizing actuals-to-formals map*(MAFM) and the corresponding value of k/N, the *maximal test frequency* (MTF).

Ranking API Methods against Sub-query *q* The MTF quantifies the *relevance* of the method. Since the same number of unit tests may not be available for every method, the *confidence* in a retrieved method does not depend on its MTF alone. For example, if two methods match a query on all their tests, but one has only 1 test while the other has 10, intuitively, the confidence in the latter is higher. We therefore normalize the number of tests per method using a constant c, by scaling the MTF by the minimum of N/c and 1. We consider a method with side-effects more difficult to use than one without, and impose a *side-effect penalty*, *sep*, on it. We set *sep* to a small positive constant for methods with side-effects and to 0 otherwise. We assign scores to methods according to:

$$\mathsf{Score}(q, m) \triangleq \min(\tfrac{N}{c}, 1).\tfrac{k}{N}.\tfrac{1}{1+sep}$$

We then rank (sort) the methods in the decreasing order of their scores (e.g. see Table 2).

Generating Pseudo-code for Expressions In general, a math expression may have many operators, with multiple candidate methods available to implement each. Consider a query Q and a set of candidate methods $\{m_1, \ldots, m_n\}$ to implement it. These are obtained by decomposing the query into sub-queries $\{q_1, \ldots, q_n\}$, and matching them as outlined earlier. The decomposition is type-correct (by construction) in the math language; however, a pseudo-code snippet to implement it must respect the type-constraints imposed by the map between library types and math types as well.

The problem is then to filter the set of all possible candidate-method sets to only those that are type-consistent, and then generate pseudo-code snippets. This can be done with an exhaustive search over the ranked lists of methods retrieved against the sub-queries. The snippets are ranked by taking the average of the scores of the methods:

$$\mathsf{Score}(Q, \{m_1, \ldots, m_n\}) \triangleq \frac{1}{n} . \sum_{i=1}^{n} \mathsf{Score}(q_i, m_i)$$

4 Unit Test Mining

In this section, we present an algorithm to compute scores of API methods against a (sub)expression containing a single operator on the RHS.

Sequential Algorithm We first present a sequential algorithm for mining unit tests (See Fig. 2).

```
Input: Query $q \equiv x = e$, unit tests of method $m$
Output: The MAFM and MTF for $m$
 1: $\Sigma \leftarrow$ set of unit tests of $m$
 2: for each $\sigma \in \Sigma$ do
 3:     Look for side-effects in $\sigma$
 4:     for each $\lambda \in \Lambda(q, m)$ do
 5:         Let $\sigma'$ be obtained from $\sigma, f$ and $\lambda$
 6:         if $\sigma'(x) = \sigma'(e)$ then
 7:             $\mathsf{Count}(\lambda) \leftarrow \mathsf{Count}(\lambda) + 1$
 8:         end if
 9:     end for
10: end for
11: Let $\lambda^*$ be such that $\mathsf{Count}(\lambda^*)$ is maximum
12: MTF $\leftarrow \mathsf{Count}(\lambda^*)/|\Sigma|$
```

Fig. 2: Sequential Mining Algorithm

As input, the algorithm takes the query q, with query expression $x = e$, and a method m (together with its unit tests). Its goal is to compute the number of unit tests that match q under every actuals-to-formals map of m. For each unit test of m (line 2), the algorithm iterates over the space of actuals-to-formals maps $\Lambda(q, m)$, and constructs a map σ' from query variables to values (in terms of the interpreter data-types, line 5). If under a particular actuals-to-formals map, the query evaluates to true (line 6) we increment a counter (the counter is initially set to 0). Finally, the algorithm returns the maximal actuals-to-formals map λ^* and the maximum test frequency. This algorithm also detects side-effects; it identifies side-effects on each method parameter (line 3) by equating the input/output values of the parameter. If there exists a test where they do not match, it sets *sep* to a small positive constant (not shown in Fig. 2).

MapReduce Version We can easily parallelize our mining algorithm. In particular, the innermost loop (over λ, line 4) can be executed over different unit tests in parallel. We exploit this data-parallelism to obtain a scalable MapReduce version of the mining algorithm.

In the MapReduce programming model [5], the input data to a *mapper* is a set of key-value pairs. The mapper's computation is applied to each key-value pair independently. It can thus be distributed over multiple nodes. After processing a key-value pair, the mapper can emit an arbitrary number of intermediate key-value pairs. These pairs represent partial results. The framework then performs a distributed group-by operation on the intermediate key, and accumulates all the values associated with it in a list. The *reducer* gets as its input the intermediate keys and the corresponding lists. It typically goes over the list of values associated with a key to compute a final result (aggregate). In a MapReduce framework, the user only has to provide implementations of the mapper and the reducer; the framework handles distribution, fault-tolerance, scheduling etc.

Our MapReduce algorithm is supported by a distributed index of unit tests. We omit details of the index organization in the interest of space. Unit tests are read from the index as the mapper's input. The mapper evaluates the subexpression on a test under every actuals-to-formals map, emitting an intermediate key-value pair $\langle \lambda, true \rangle$ or $\langle \lambda, false \rangle$ for each. This partial result says whether the subexpression evaluated to true or false under a particular λ. After the runtime performs a distributed group-by operation, a key-value pair arriving at a reducer contains an actuals-to-formals map λ (key) and a list of booleans (value). Every entry in this list was generated by evaluating λ on some unit test. Values of k and N are calculated for a particular λ by iterating over this list. λ^* (the maximizing actuals-to-formals map) is the key that maximizes k/N over all key-value pairs. In our implementation, we lift this algorithm to work on indices containing unit tests of multiple methods from across libraries.

5 Implementation and Evaluation

Implementation We implemented the MapReduce version of the mining algorithm in the Apache `Hadoop` framework, with Scilab as the interpreter. We ran the mining algorithm on a unit test index containing unit tests from our target libraries, that we wrote using JUnit. We used `Serialysis`[6] to serialize input/output values from the unit tests, and provided hooks in our framework for specifying the mapping between library and math types. As an optimization, we cached the top k methods retrieved against every operator in the interpreted language in an *operator index*, which is a Java HashMap, serialized to disk.

We implemented MathFinder as an Eclipse plugin that interfaces Eclipse with the API discovery and snippet-generation engines. In Eclipse, the Math-

[6] weblogs.java.net/blog/emcmanus/serialysis.zip

Finder view offers a search bar to type math expressions in. We parse, type-check and decompose expressions into subexpressions using the `Antlr3`[7] framework. Subexpressions are answered in real-time by looking up the operator index.

User Study We conducted a user study to measure whether MathFinder improves programmer productivity on mathematical programming tasks when compared to reading Javadoc, using Eclipse code-completion, and keyword-driven web or code search. To measure this, we picked a set of four mathematical programming tasks (see Table 3 for a summary) that required third-party libraries to complete. Only the first task could be implemented using any target library, while the others required a careful evaluation of the APIs to find a functionally complete target library. We presented the algorithms to the participants as method stubs in Eclipse.

Table 3: Summary of the tasks used in the user study

Task	Algorithm Name	Description
1	Conjugate Gradient	Linear Equation Solving
2	Chebyshev	Polynomial Interpolation
3	PageRank	Webpage Ranking
4	Rayleigh Iteration	Eigenvalue Computation

We deemed participants to have completed a task when their program passed all our unit tests. The main barrier to implementation was the lack of direct Java support, rather than algorithmic subtleties. We chose small tasks, expecting the participants to finish them within two hours. There were 16 unique operators across the tasks, and 5 to 8 queries in each task whose implementation required method composition. The target libraries were Colt, EJML, Jama and JBlas.

Our participants were 5 industry professionals and 3 graduate students not affiliated to our research group. Two participants attempted every task, one without MathFinder and one with MathFinder. Those in the *control group* were allowed to use Javadoc, Eclipse code completion, and web or code search engines; in addition, those in the *experimental group* were allowed to use MathFinder. We gave the participants handouts describing the operators used in the tasks, and a mapping between library types and math types (similar to Table 1).

Timing Results Table 4 shows the timing results of the user study. All times are in minutes. MathFinder users finished 1.96 times as fast as the control-group participants on average. Though the study is not large enough to measure the difference with statistical significance, these results suggest that MathFinder helps improve productivity.

Table 4: Task completion times

Task	Control group	Experimental group(speed-up)
1	95m	51m(1.86x)
2	93m	64m(1.45x)
3	97m	39m(2.49x)
4	75m	30m(2.50x)

Control group participants reported that they found selecting an API that best supported their task difficult, often requiring a search over Javadoc pages of multiple libraries. We expected participants to pick keywords out of operator descriptions in the handout (e.g., "matrix multiplication") and use Google or code search en-

[7] antlr.org

gines, but surprisingly, only one participant did so. This may be due to the difficulty of analyzing a number of independent search results, pertaining to individual operators in the task. Almost all control group users relied on Eclipse code completion. This proved unhelpful at times, given the sheer number of methods in some relevant API classes, similar names, and because the required functionality was spread across multiple classes. For example, there are at least 16 methods in the `CommonOps` class of `EJML` with a prefix "mult", and all have something to do with matrix multiplication. `JBlas` has 19 such methods in the `DoubleMatrix` class. Although we did not provide type-based API discovery tools [16, 24] to aid participants, we believe that these would not have altered the outcome significantly. Type-based queries can result in many spurious results for math APIs because a large number of methods operate over the same types. For example, the JBlas library has over 60 methods that take two `DoubleMatrix` objects as input and return a `DoubleMatrix` object. Searching by method signatures cannot distinguish, say, matrix addition from matrix multiplication.

MathFinder users, on the other hand, were able to formulate queries directly from the tasks, and all of them reported that the tool was easy to use. With the tool, they were able to quickly gauge the extent of library support for their task across libraries, and zero-in on the right library. The queries returned precise results, and usually, the participants did not have to look beyond the top ranked snippet. They copied the suggested snippets into the workspace and completed them, consulting the Javadocs only to find appropriate constructors. This experience leads us to believe that MathFinder will deliver larger productivity gains with more complex tasks and diverse API requirements.

Precision and Recall We evaluate the precision of our approach on both API discovery and synthesis of pseudo-code snippets. To evaluate precision of API discovery, we picked the set of unique operators from across the tasks; there were 16. Of these, Colt supports 7, EJML 13, Jama 13 and JBlas 14. For unsupported operators, MathFinder returns empty results, since it picks only results with a score above a threshold (0.75). The precision on operators supported by individual libraries is given in Table 5. The precision is high (96% on average), despite the fact that these libraries use different class definitions, calling conventions, etc. Also, MathFinder retrieved all relevant methods from all libraries (recall 1), with two exceptions. One was the `eyes` operator, used to generate identity matrices. The corresponding JBlas method `eyes` was not retrieved. This method takes only one argument (equal to both the number of rows and columns), whereas the `eyes` operator in Scilab takes two integer arguments (rows and columns) separately. A relaxed type matching may help us identify methods like `eyes` that take fewer parameters than the subexpression variables. The other exception was the transpose operator. MathFinder mapped it to an

Table 5: Precision of API Discovery

Library	$\frac{\text{\#correct@rank-1}}{\text{\#supported-operators}}$
Colt	6/ 7(86%)
EJML	13/13(100%)
Jama	13/13(100%)
JBlas	13/14(93%)
Total	45/47(96%)

incorrect method of Colt. Later, we were able to attribute it to having missed a special case in mapping library datatypes to interpreter datatypes. This implementation issue was easy to fix, but we only report results prior to the fix.

There were 24 expressions in total in all the tasks. The operators used in these expressions were not supported by every library. Therefore, to measure the precision of pseudo-code snippet synthesis on a library, we only considered expressions that could be implemented fully using it. With this restriction, Colt supports 6 expressions, EJML 17, Jama 15, and JBlas 17. For the expressions that could be implemented, we evaluated, for each library, whether the top-most code snippet MathFinder returned was correct. The results are given in Table 6. The precision across libraries is 93% on average. Our technique is able to mine operator to method maps as well as maps from actuals to formals accurately, which in turn means that the synthesized pseudo-code snippets are precise. Colt's precision was low because 4/6 expressions used transpose (which was mapped to an incorrect method).

Table 6: Precision of synthesized pseudo-code snippets

Library	$\frac{\text{\#correct-snippet}}{\text{\#expressions}}$
Colt	2/ 6(33%)
EJML	17/17(100%)
Jama	15/15(100%)
JBlas	16/16(100%)
Total	50/54(93%)

Threats to Validity Threats to *internal validity* include *selection bias* where the control and experimental groups may not be equivalent at the beginning of the study, and *testing bias* where pre-test activities may affect post-test outcomes. To prevent selection bias, we conducted a survey before the study and paired programmers with similar levels of Java expertise. We then assigned them the same task, but chose their group (control or experimental) randomly. To mitigate testing bias, we gave the experimental group participants a 20 minute presentation on the tool instead of a hands-on tutorial. Threats to *external validity* arise because our results may not generalize to other groups of programmers and programming tasks. To ensure a level playing field, we made sure none of our participants had prior exposure to the target APIs. But this meant that we had to leave out expert users of the target APIs. Therefore, the study does not assess the benefits of MathFinder to domain experts and whether selecting another candidate library is as difficult for them as for programmers with no experience with any of the libraries. As target APIs, we picked popular open-source third-party libraries which we believe are representative. However, further studies are needed to validate the findings for other APIs in the math domain.

Scalability and Response Time Our retrieval target collection had 406 methods: 41 methods from Colt, 70 from EJML, 45 from Jama, and 250 from JBlas. We obtain the time for API discovery using 10, 200 and 500 tests/method against queries involving operators used in the tasks. With 10 tests/method, the experiments were performed on a desktop running Ubuntu 10.04 with an Intel i5 CPU (3.20GHz, 4GB RAM). We used a single mapper and reducer to run the MapReduce implementation of the mining algorithm. With 200 and 500 tests/method, the experiments were carried out on a machine running CentOS 5.3, with 8 Xeon quad-core processors (2.66GHz, 16GB RAM). The machine could run up to 7 mappers and 2 reducers. For computing scores, we set the side-effect-penalty to 0.2. The processing time per query was 3.7s on average with 10 tests/method (desktop), 56.7s with 200 tests/method and 80.5s with 500 tests/method (multi-core processor). The precision of results did not vary significantly with 200 or 500 tests/method, suggesting that for the domain we considered, our technique is able to achieve high precision with only a few tests/method. The study shows that our implementation scales to an index with over 200K test records.

Limitations Our ranking function does not take into account performance or efficiency of API implementations. It does not rank an API method that is a specialization of the math operator lower. We cannot discover compositions of API methods to implement a single operator. Our approach, in its current form, cannot discover APIs that take function objects as parameters, e.g., one of our target libraries, Colt, has a set of functions available through a function `assign` which takes function objects as input. The equality between query and method outcomes is relaxed, in that, double precision numbers computed by the interpreter and by an API method are said to be equal if they are within $\epsilon = .001$ of each other. The incompleteness or errors in data (unit tests) can affect precision of results. This is true of any data mining approach.

6 Related Work

In text search, the popularity of web search engines shows that keyword-driven queries are used extensively. In programming, the main utility of web search engines seems to be to retrieve library documentation. Commercial code search engines (e.g. Codase, Google code search, Koders, Krugle, etc.)[8] retrieve declarations and reference examples given library and method names. These approaches are difficult to use if the programmer does not know the suitable libraries or methods to begin with. Some research tools like Assieme [9], Codifier [2], and Sourcerer [14] can perform syntactic search using richer program structure. The MathFinder approach is purely *semantic* and does not use keywords or program structure for search.

Several approaches [22, 28, 16, 24] use types for API discovery. These approaches discover API call sequences to go from an input type to an output type by mining API declarations and in some cases, client source code. A dynamic analysis approach, MatchMaker [27], discovers API sequences by mining program traces. In our experience, the objects set up using math APIs are easy to initialize and do not require a sequence of calls to set up state before they may be used. Since types alone may not be enough for accurate API discovery, some techniques combine them with structural contexts including comments, field/method names, inheritance relations, and method-parameter, return-type, and subtype relations [26, 10, 23, 6]. Types are also combined with keywords in Keyword Programming [15] and SNIFF [3]. Apart from APIs, type-based code completion approaches such as InSynth [7] and the work of Perelman et al. [19] also search over variables in the typing context.

[8] respectively, codease.com, code.google.com/hosting, koders.com, krugle.org

The main limitations of type-driven approaches include (i) the assumption that the programmer has (partial) knowledge of the types and (ii) the lack of precise semantic information in the queries. In MathFinder, the programmer formulates queries over mathematical types (of the interpreted language used) and not over library types. Thus, the same query is enough to discover APIs across multiple libraries. Our queries are math expressions over interpreted operators and can accurately identify methods for the operators in the query expression.

Prime [17] queries are partial programs, from which it mines partial temporal specifications and matches them against an index of temporal specifications built from example code from the web. We have already compared our work, in Section 1, with more closely related approaches like specification-driven [29, 21] and test-driven [20, 8, 11, 21, 13] techniques for API discovery.

7 Conclusions and Future Work

This paper presents a novel technique to search for math APIs. A programmer submits a math expression directly as a query to MathFinder which returns pseudo-code for computing it by composing library methods. The approach combines executable semantics of math expressions with unit tests of methods to mine a mapping from expression variables to method parameters and detects likely side-effects of methods. We show that the approach improves programmer productivity, gives precise results, and scales to large datasets.

The availability of rigorous specifications make mathematical computations an attractive choice for automated code synthesis. The existence of mature libraries makes the synthesis problem in this domain more about API discovery than algorithm discovery. Our work is a step toward API-driven synthesis.

Some methods may take more parameters than the corresponding math operator. Mining initializations to these parameters from unit tests is an interesting future direction. We also plan to explore more general queries involving predicates. API migration is a potential application of our unit test mining approach. The semantics of APIs to be migrated can be specified in math notation, to obtain matching APIs from other libraries using MathFinder.

Acknowledgements We thank the volunteers of the user study, and the members of the Software Engineering and Analysis Lab for their help during the user study. We also thank the anonymous reviewers for their valuable feedback.

References

1. A.V. Aho, M. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Prentice Hall, 2006.
2. A. Begel. Codifier: a programmer-centric search user interface. In *Workshop on Human-Computer Interaction and Information Retrieval*, 2007.
3. S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A search engine for Java using free-form queries. In *FASE*, pages 385–400, 2009.

4. B.N. Datta. *Numerical Methods for Linear Control Systems*. Elsevier Inc., 2004.
5. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
6. E. Duala-Ekoko and M. P. Robillard. Using structure-based recommendations to facilitate discoverability in APIs. In *ECOOP*, pages 79–104, 2011.
7. T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. In *CAV*, pages 418–423, 2011.
8. R. J. Hall. Generalized behavior-based retrieval. In *ICSE*, pages 371–380, 1993.
9. R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *UIST*, pages 13–22, 2007.
10. R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE*, pages 117–125, 2005.
11. O. Hummel, W. Janjic, and C. Atkinson. Code Conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25(5):45–52, 2008.
12. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188, 1999.
13. O. A. L. Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Inf. Softw. Technol.*, 53(4):294–306, April 2011.
14. E. Linstead, S. K. Bajracharya, T. C. Ngo, P. Rigor, C. V. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.*, 18(2):300–336, 2009.
15. G. Little and R. C. Miller. Keyword programming in Java. In *ASE*, pages 84–93, 2007.
16. D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI*, pages 48–61, 2005.
17. A. Mishne, S. Shoham, and E. Yahav. Typestate-based semantic code search over partial programs. In *OOPSLA*, pages 997–1016, 2012.
18. L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.
19. D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *PLDI*, pages 275–286, 2012.
20. A. Podgurski and L. Pierce. Behavior sampling: a technique for automated retrieval of reusable components. In *ICSE*, pages 349–361, 1992.
21. S. P. Reiss. Semantics-based code search. In *ICSE*, pages 243–253, 2009.
22. M. Rittri. Retrieving library identifiers via equational matching of types. In *CADE*, pages 603–617, 1990.
23. N. Sahavechaphan and K. Claypool. XSnippet: mining for sample code. In *OOPSLA*, pages 413–430, 2006.
24. S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE*, pages 204–213, 2007.
25. J. Vesanto, J. Himberg, E. Alhoniemi, and J. Parhankangas. Self-Organizing Map in Matlab: the SOM toolbox. In *Proc. of the Matlab DSP Conf.*, pages 35–40, 2000.
26. Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *ICSE*, pages 513–523, 2002.
27. K. Yessenov, Z. Xu, and A. Solar-Lezama. Data-driven synthesis for object-oriented frameworks. In *OOPSLA*, pages 65–82, 2011.
28. A. M. Zaremski and J. M. Wing. Signature matching: a key to reuse. In *FSE*, pages 182–190, 1993.
29. A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.*, 6(4):333–369, October 1997.