# Mining Unit Tests for Discovery and Migration of Math APIs

ANIRUDH SANTHIAR, Indian Institute of Science
OMESH PANDITA, Indian Institute of Science
ADITYA KANADE, Indian Institute of Science

Today's programming languages are supported by powerful third-party APIs. For a given application domain, it is common to have many competing APIs that provide similar functionality. Programmer productivity therefore depends heavily on the programmer's ability to discover suitable APIs both during an initial coding phase, as well as during software maintenance.

The aim of this work is to support the discovery and migration of math APIs. Math APIs are at the heart of many application domains ranging from machine learning to scientific computations. Our approach, called MATHFINDER, combines executable specifications of mathematical computations with unit tests (operational specifications) of API methods. Given a math expression, MATHFINDER synthesizes pseudo-code comprised of API methods to compute the expression by mining unit tests of the API methods. We present a sequential version of our unit test mining algorithm and also design a more scalable data-parallel version.

We perform extensive evaluation of MATHFINDER (1) for *API discovery*, where math algorithms are to be implemented from scratch and (2) for *API migration*, where client programs utilizing a math API are to be migrated to *another* API. We evaluated the precision and recall of MATHFINDER on a diverse collection of math expressions, culled from algorithms used in a wide range of application areas such as control systems and structural dynamics. In a user study to evaluate the productivity gains obtained by using MATHFINDER for API discovery, the programmers who used MATHFINDER finished their programming tasks twice as fast as their counterparts who used the usual techniques like web and code search, IDE code completion, and manual inspection of library documentation. For the problem of API migration, as a case study, we used MATHFINDER to migrate Weka, a popular machine learning library. Overall, our evaluation shows that MATHFINDER is easy to use, provides highly precise results across several math APIs and application domains even with a small number of unit tests per method, and scales to large collections of unit tests.

Categories and Subject Descriptors: D [**Software**]: Automatic Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.13 [**Software Engineering**]: Reusable Software

General Terms: Programming, Algorithms, Design, Experimentation

Additional Key Words and Phrases: API discovery, API migration, mathematical computation, mining, unit tests

## 1. INTRODUCTION

Modern programming languages come equipped with rich APIs. In addition, commercial and open-source organizations often enable code-reuse via their own APIs. Given the ubiquity of API usage in software development today, programmer productivity depends heavily on the programmer's ability to discover the required APIs and to learn

Author's addresses: A. Santhiar and A. Kanade, Department of Computer Science and Automation, Indian Institute of Science; O. Pandita, now with Microsoft India Development Center.

DOI: http://dx.doi.org/10.1145/0000000.0000000

to use them quickly and correctly. While API discovery and comprehension is critical to building new components using the API, it also plays an important role in many software maintenance activities such as upgrading to a newer version of an API or migrating to a different API. But having to choose from competing APIs, given their sizes and the varying support they provide for computational tasks, imposes a significant burden on programmers. A programmer has to discover and learn to use the methods that provide a particular functionality, and compare and contrast multiple APIs, gauging how well they support the programming task at hand.

Significant research efforts are targeted at aiding programmers in API discovery. A programmer can search for APIs using a wide spectrum of techniques. They range from keywords [Little and Miller 2007; Chatterjee et al. 2009], types [Mandelin et al. 2005; Thummalapenta and Xie 2007], tests [Hummel et al. 2008; Lemos et al. 2011], and code snippets [Mishne et al. 2012], to formal specifications [Zaremski and Wing 1997] or combinations of the above [Reiss 2009]. 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).

Existing research has also addressed the migration of a client from a *source* API to a *target* API. Many tools like CatchUp [Henkel and Diwan 2005], the tool designed by Schäfer et al. [Schäfer et al. 2008], SemDiff [Dagenais and Robillard 2009], LibSync [Nguyen et al. 2010], AURA [Wu et al. 2010] and HiMa [Meng et al. 2012] mine mappings from source API elements to target API elements for version-related APIs. Other works [Zhong et al. 2010; Gokhale et al. 2013] propose techniques to mine these mappings automatically in a setting where the source and target APIs are not version related. However, these approaches require other clients that have already been ported correctly from the source API to the target API.

In this paper, we address the problem of API discovery and migration for mathematical computations. Mathematical computations are at the heart of numerous application domains such as statistics, machine learning, image processing, engineering or 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 [Vesanto et al. 2000; Datta 2004]). 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 issues 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.

We present a *unified* approach to assist programmers in the discovery and migration of math APIs. In the *API Discovery* problem, the programmer is given a new algorithm to implement, specified in the notation of an interpreted language. She must identify an API that supports most of the computations involved, as well as methods within the API to perform the computations. Our technique, called MATHFINDER, permits the programmer to pose individual math expressions from the algorithm as *queries*. For example, the programmer may query *double M* v; v = v ./ normf(v)

with *double M* v indicating that v is a matrix of doubles, and normf standing for the Frobenius norm, and ./ for matrix-scalar division. In response to this query, MATHFINDER returns the following *pseudo-code snippet* comprised of methods from the JBlas[1] API:

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

As a notational convention, temporary variables are given names T1, T2, etc. in the pseudo-code snippets. The names of the other variables match the names of the corresponding variables in the query.

In the *API migration* problem, the programmer is required to migrate a client of an API $X$ to another API $Y$ (possibly, but not necessarily a newer version of $X$). Consider the following statement taken from a client of the Jama[2] matrix library.

```
tmp = W.times(((P.transpose()).times(W)).inverse());
```

The programmer can query MATHFINDER in an *API independent* manner by expressing the computation in the above statement as a math expression:

*double M* tmp, W, P; tmp = W*inv(P'*W);

In response to this query, MATHFINDER returns the following pseudo-code snippet based on the EJML[3] API, that can be used to migrate the statement above to a semantically equivalent sequence of statements that use EJML.

```
SimpleMatrix tmp, P, W, T1, T2, T3;
T1 = P.transpose(); T2 = T1.mult(W);
T3 = T2.invert(); tmp = W.mult(T3);
```

In order to synthesize the pseudo-code snippets given in the examples above, MATHFINDER must infer which API calls implement the computation equivalent to a subexpression in the query, and what the mapping between the query variables and the method parameters must be. For example, in the first query above, MATHFINDER identifies the method `double DoubleMatrix.norm2()` as suitable for computing the subexpression normf(v). It infers that v should be mapped to `this` and that the result is available in the return value of the method call (in general, the result could be available in some parameter to the method). MATHFINDER also discovers if a method is likely to modify the input parameters (i.e., it discovers likely side-effects of methods). If there is a side-effect on some parameter, then the pseudo-code must clone that parameter into a temporary before the call, in case it is used in a subsequent subexpression. Obtaining all this information is a challenging problem.

Formal specifications of semantics of methods may help us solve this problem. Languages like JML [Leavens et al. 1999] 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[4]. We therefore use (the set of input/output objects in) unit tests as a description of method semantics. While we chose Java as the target programming language, our technique can work with other languages as well.

[1]jblas.org
[2]math.nist.gov/javanumerics/jama/
[3]code.google.com/p/efficient-java-matrix-library/
[4]junit.org

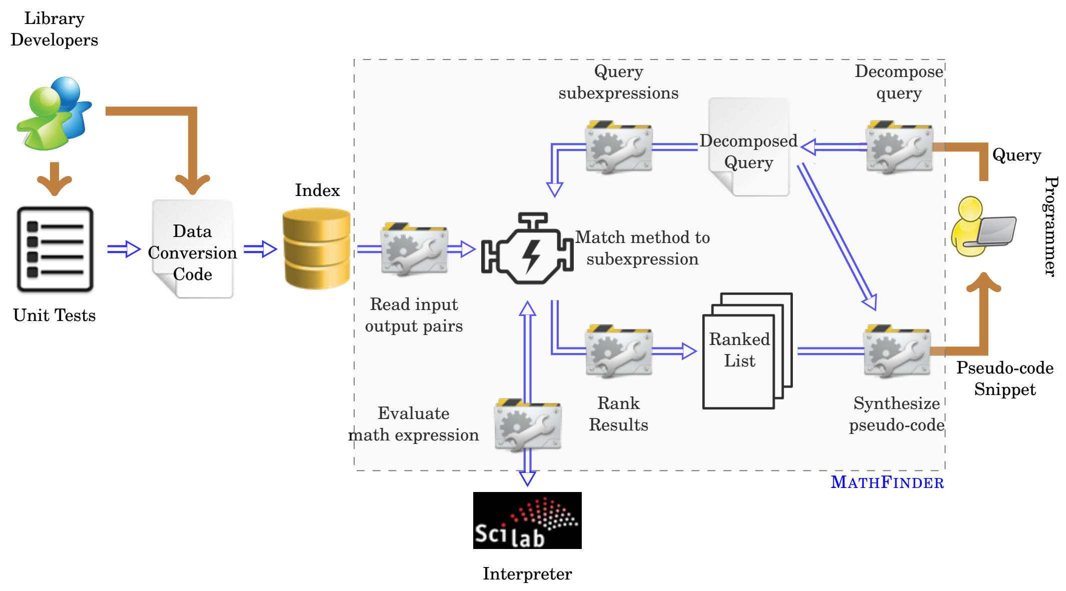

Fig. 1: High-level view of the MATHFINDER approach

### 1.1. The MATHFINDER Approach

The key insight in MATHFINDER is to use an interpreter for a math language (such as Scilab[5]) to evaluate subexpressions on unit tests of library methods. The result of interpreting a subexpression 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. However, a subexpression cannot be directly evaluated on data from unit tests because the math types used in the expression are *independent* of the data-types used in library APIs. We therefore require the library developer to provide code to convert library data-types to types of the math interpreter. As shown in Fig. 1, library developers make their libraries amenable to our technique by providing unit tests, and plugging data-type conversion code into our framework. In the case of the example shown earlier to illustrate API discovery, 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 data-types to interpreter data-types is a one time task and was straight-forward in our experience.

Given a query by the programmer, MATHFINDER decomposes the query and extracts subexpressions from it (Fig. 1, top right). 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 data-types 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`, obtained from the unit tests, to v. Then, normf(v) evaluates to a value equal to the return value of `norm2()` on every unit

[5]scilab.org

test of `norm2()` supplied. This evaluation is performed by the interpreter of the math language. Alongside, MATHFINDER also infers likely *side-effects* of a method call by contrasting the values of method parameters at input and output. As shown in Fig. 1, MATHFINDER ranks the methods based on the frequency of unit tests that match a subexpression. It constructs a ranked list of matching methods obtained against every subexpression. Finally, it synthesizes pseudo-code using the decomposed query and the ranked lists of methods. It may synthesize multiple candidate pseudo-code snippets, and it ranks them based on a score representing their confidence and ease of use.

Our approach falls in the category of specification-driven API discovery [Zaremski and Wing 1997; Reiss 2009]. 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 data-types to the math types. In contrast, in test-driven API discovery approaches [Podgurski and Pierce 1992; Hall 1993; Hummel et al. 2008; Reiss 2009; Lemos et al. 2011], 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 data-types and invest time in writing unit tests. In our approach, the programmer query is independent of library data-types (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.

## 1.2. Results

We evaluated the precision and recall of MATHFINDER on a large and diverse collection of math expressions, culled from algorithms used in a wide range of application areas such as control systems and structural dynamics. The precision of pseudo-code synthesis on this set of benchmark queries was 98%. We also queried for a large number of operators used in these algorithms. The API method retrieved as the top-most result against such a subexpression query was correct 91% of the time. High precision was obtained using only 10 tests/method.

We conducted a user study to evaluate the productivity gains obtained by using MATHFINDER for API discovery. The programmers who used MATHFINDER finished their programming tasks twice as fast as their counterparts who used the usual techniques like web and code search, IDE code completion, and manual inspection of library documentation. MATHFINDERS's results were quite precise across multiple libraries for the tasks used in the study. 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.

The unit test mining approach of MATHFINDER provides useful information in terms of the use of method parameters and method side-effects. Such information is usually available in the API documentation. During the course of evaluating MATHFINDER, we found discrepancies between MATHFINDER's output and the Javadoc of the JBlas library. While MATHFINDER indicated no side-effect on some methods, their documentation explicitly states that they perform computations "in-place"[6], i.e., a side-effect is expected on the `this` parameter. We studied the method implementations and found that the documentation was indeed inaccurate; the methods had no side-effects.

For the problem of API migration, as a case study, we used MATHFINDER to migrate a popular machine learning library, Weka, to matrix and linear algebra libraries that are more efficient and well-supported than the Jama library it uses currently. The

[6]jblas.org/javadoc/index.html, e.g., the `add` method

migration task spanned 20 classes and 116 methods. In 94% of the queries, MATHFINDER returned correct pseudo-code snippets from the target APIs. As we discuss later, not all migration sites are amenable to our technique, e.g. the type declarations and object creations involving the source API. These program statements need to be migrated manually. In our case study, we were able to handle 51% of the migration sites with MATHFINDER. Using MATHFINDER, we could also exploit opportunities for reuse where some client methods were supported by the target libraries and could be eliminated. We could also migrate the program statements that still used some deprecated classes of Weka. We validated the correctness of the migrated versions of Weka using its extensive regression test suite. All the tests except for one passed on our migrated versions.

We show that our technique can be effectively parallelized. Since the test suite collection can be large in practice, we also implemented it in the Hadoop[7] MapReduce framework. It scaled to a large collection of unit tests consisting of over 200K tests and returned results in 80.5s on average, on an 8-core machine. These results are cached for real-time retrieval using the Eclipse plugin.

Overall, our evaluation shows that MATHFINDER is easy to use, provides highly precise results across several math APIs and application domains even with a small number of unit tests per method, and scales to large collections of unit tests.

## 1.3. Contributions

A preliminary description of the MATHFINDER approach was published in [Santhiar et al. 2013]. Compared to it, we make the following new contributions in this article:

(1) We propose an approach to assist programmers in math API migration. We undertake a large case study of migration of a machine learning library to demonstrate the applicability of the unit test mining approach. This extends the applicability of MATHFINDER from its use in API discovery presented in [Santhiar et al. 2013].
(2) We present the data-parallel version of the unit test mining algorithm and describe how the algorithm is implemented over a cluster and a distributed file system. In a real-world setting, we expect a tool like MATHFINDER to be used with a large number of APIs and consequently, a large number of unit tests. We demonstrate the scalability of our approach with the data-parallel algorithm.
(3) The user study pertaining to API discovery presented in [Santhiar et al. 2013] is relatively small. We augment that evaluation with an empirical study involving a large benchmark of queries.
(4) We have released the source code of the MATHFINDER Eclipse plugin, and the sequential and data-parallel mining algorithms. All the problems used in the user study, the unit-tests of the libraries indexed, the benchmark queries, and the migrated versions of Weka are publicly available at http://www.iisc-seal.net/mathfinder.

**Organization** We present examples to illustrate the MATHFINDER approach in the next section. We discuss the algorithms and implementation in Sections 3 - 4, and evaluate MATHFINDER in Sections 5 - 7. We discuss additional aspects of our technique and its limitations, and sketch future work in Section 8. We survey related approaches in Section 9. Finally, we conclude in Section 10.

[7]hadoop.apache.org

## 2. ILLUSTRATIVE EXAMPLES

In this section, we illustrate the individual steps of the MATHFINDER approach with two examples derived from real-world programming and migration tasks.

### 2.1. Implementing the PageRank Algorithm

Let us first consider the task of implementing a math algorithm from scratch. Fig. 2 shows the PageRank algorithm used by Google for ranking web pages. The algorithm is presented in the notation of Scilab.

```
% 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. 2: Scilab code for PageRank (adapted from Wikipedia)

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. Of the libraries we surveyed, namely Colt [8], EJML, Jama and JBlas, only JBlas (containing over 250 methods for matrix operations) supports all the required operators. The programmer must identify that JBlas is the right library to implement this algorithm. 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 expressions from the algorithm to query MATHFINDER for APIs to implement them. 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. We now describe the individual steps involved in returning a pseudo-code snippet to the user.

*Interpreter data-types*. Suppose the programmer wants to find out how to implement the assignment in line 6, v = v ./ normf(v) (discussed earlier in Section 1). We use the math types *double* (*int*) standing for double (integer) scalars, and *double M*, for double matrices. Variable v is typed with *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. As discussed in Section. 1, for each library, its developer provides a mapping between math

| Library | Class |
|---|---|
| Colt | `DenseDoubleMatrix2D` |
| EJML | `DenseMatrix64F, SimpleMatrix` |
| Jama | `DoubleMatrix` |
| JBlas | `Matrix` |

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

[8]acs.lbl.gov/software/colt/

| 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 |

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

types and the classes used in her library, along with code for converting the library's objects to values of the math type.

Table I lists classes across four math libraries that we use as targets for API discovery. This helps us translate type signatures and data between library types and math types. Thus, the queries themselves are independent of the target libraries. That is, the same query suffices to search over multiple libraries. MATHFINDER exports the operators defined in the interpreted language for the programmer to use in queries. It also permits programmers to add support for new operators by defining them using the syntax of the interpreted language.

*Query Decomposition*. MATHFINDER parses the math expression and decomposes it into subexpressions (similar to three-address code generation in compilers [Aho et al. 2006]). 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 (by disabling query decomposition).

*Unit Test Mining*. 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 data-types (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 II. In the actuals-to-formals maps in Table II, `arg1` stands for the first formal (argument), and `return` stands for the return value of the method. In other words, when values of `arg1` from the tests were placed into v, the result of evaluating normf(v) agreed with the values of `return` from the corresponding tests. Even though multiple methods with score 1.0 are retrieved, they come from different libraries (refer Table II) and any one of them can be used to implement the queried subexpression. 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 I.

*Ranking Results*. 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

```
// W*(P^T*W)^-1
tmp = W.times(((P.transpose()).times(W)).inverse());

// X_new = X*W*(P^T*W)^-1
X_new = getX(instances).times(tmp);

// factor = W*(P^T*W)^-1 * b_hat
m_PLS1_RegVector = tmp.times(b_hat);
```

Fig. 3: Code from the `PLSFilter` class of Weka

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 II. 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.

*Synthesizing pseudo-code snippets*. 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 would be compatible, and the output object is the return value of the last method. MATHFINDER suggests this snippet from JBlas for our running example.

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

Here `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.

### 2.2. Migrating a Class Implementing Partial Least Squares Regression

We now consider the problem of API migration with an example taken from a popular machine learning library, Weka [Hall et al. 2009]. Consider the code snippet from the `PLSFilter` class of Weka, shown in Fig. 3. This code is a part of a method that implements the PLS1 algorithm to find the Partial Least Squares Regression on a given data set (the details of this algorithm are not important for this discussion). The developers of Weka have used the operators inverse, times and transpose in this snippet. Weka uses their Jama implementations. Jama saw its last release in 2005 and is not under active development. A number of libraries with functionality similar to Jama but with better performance and support are available now [Abeles 2010]. Weka is therefore an ideal candidate for API migration.

To perform the migration using our approach, the first step would be to frame a math query that embodies the semantics of the statement to be migrated. This involves mapping the library methods used in the statement to operators of the math expression language. We note that mathematical expressions explaining what a statement or a sequence of statements does are sometimes explained in comments. The developer can make use of these comments to frame the query. In particular, the comments shown in Fig. 3 can be used as queries to MATHFINDER. We note that these comments already exist in the current implementation of the `PLSFilter` class. The following query asks for candidate snippets from target libraries:

*double M* tmp, W, P; tmp = W$*$inv(P'$*$W);

Apparently, the original library developer used '*', to denote matrix multiplication in the comments, which maps to the query operator $*$, where as the operator '^T' maps to ', and the operator '^-1' maps to inv. This query is used to fetch pseudo-code snippets from multiple target libraries. It returns, for example, the following pseudo-code snippet that uses the `SimpleMatrix` class of EJML.

```
SimpleMatrix tmp, P, W, T1, T2, T3;
T1 = P.transpose(); T2 = T1.mult(W);
T3 = T2.invert(); tmp = W.mult(T3);
```

As noted earlier in this section, the programmer should write code for object instantiation, and can potentially eliminate certain temporaries by chaining methods on the same receiver.

MATHFINDER can assist the programmer in mapping methods in the source library to equivalent methods in the target library. The number of methods of the source library that are supported by each potential target library provides an estimate of the difficulty of the migration task, and guides the choice of the target library. While this is an important step, full fledged migration involves many other issues. In addition to migrating methods, field and method declarations that use types from the source library have to be migrated to the new types from the target library. Equivalent constructors for these new types must be identified. The programmer has to supply any functionality that the source library provided, but is missing from the target. In this work, we restrict the scope to discovering equivalent methods. A common problem in API migration is that the functionality of a class of the library is spread over multiple classes in a target library. MATHFINDER relieves the programmer from manually identifying such cases. We demonstrate this point concretely in the evaluation section (Section 7).

*Querying for Reuse Opportunities*. Migrating to another third party library can lead to additional opportunities for reuse. The client code may have implemented functionality that the source library did not provide, but that is present in the target library. Our technique can be used to detect such methods in client code, as we demonstrate in Section 7.

## 3. UNIT TEST MINING

Unit test mining lies at the heart of our technique to discover APIs. It takes as inputs unit tests and maps from API data-types to interpreter data-types, supplied by the library developer, and a math subexpression queried for by the user. It outputs a ranked list of API methods that match the subexpression.

### 3.1. Problem Statement

Now, we define the problem 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. Recall that an actuals-to-formals map is a mapping between variables of the subexpression and method parameters.

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). For the subexpression T1 = normf(v) from our running example, {(v, `this`), (T1, return)} is the actuals-to-formals map for the `double Matrix.normF()` method from the Jama library.

Let a unit test $\sigma$ 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 $\sigma$ of a method $m$ and an actuals-to-formals map $\lambda$, for a subexpression $x = e$, $\sigma'$ gives the values of variables occurring in $x = e$. For a variable $y$ appearing in $x = e$,

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

The mapping $\sigma'$ 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 $\sigma$, under an actuals-to-formals map $\lambda$, 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 $\lambda$. The problem is then to find an actuals-to-formals map $\lambda^*$ that maximizes $k/N$. We call $\lambda^*$ the *maximizing actuals-to-formals map* and the corresponding value of $k/N$, the *maximal test frequency*.

*Ranking API Methods against Sub-query* $\boldsymbol{q}$. We can now define the problem of ranking API methods against a sub-query $q$. The maximal test frequency 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 maximal test frequency alone. For example, if two methods match a query on all their tests, but the former has only $1$ test, while the latter 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 maximal test frequency 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. MATHFINDER dynamically detects whether or not a method has side effects, as discussed in Section 1. 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 II).

*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

```
Input: Query $q \equiv x = e$, unit tests of method $m$
Output: The maximizing actuals-to-formals map and maximal test frequency for $m$
1 $\Sigma \leftarrow$ set of unit tests of $m$
2 forall $\sigma \in \Sigma$ do
3     Look for side-effects in $\sigma$
4     forall $\lambda \in \Lambda(q, m)$ do
5         Let $\sigma'$ be constructed using $\sigma, f$ and $\lambda$
6         if $\sigma'(x) = \sigma'(e)$ then
7             $\mathsf{Count}(\lambda) \leftarrow \mathsf{Count}(\lambda) + 1$
8 Let the maximizing actuals-to-formals map be such that $\mathsf{Count}(\lambda^*)$ is maximal
9 maximum test frequency $\leftarrow \mathsf{Count}(\lambda^*)/|\Sigma|$
```

**ALGORITHM 1:** Sequential mining algorithm

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:

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

### 3.2. Sequential Algorithm

We first present a sequential algorithm for mining unit tests (see Algorithm 1). 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 $\sigma'$ 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 $\lambda^*$ 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 Algorithm 1).

### 3.3. MapReduce Version

We now explain how to parallelize our mining algorithm. In particular, we observe that the outermost loop (over $\sigma$, line 2 of Algorithm 1) can be executed by reading 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 [Dean and Ghemawat 2008], the input data to a *mapper* is a set of key-value pairs, stored in a distributed index. 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 an aggregate (final result). 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.

Fig. 4 shows the high level architecture of our approach. We begin with unit tests supplied by the developer. After converting the data-types used in these to those used

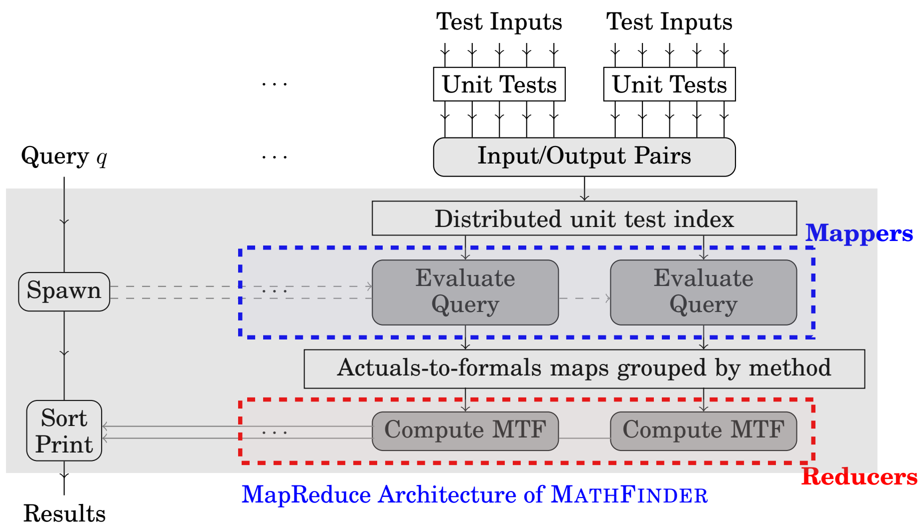

Fig. 4: Architecture of MathFinder 's data-parallel unit test mining engine

in the interpreter, we organize them into a distributed unit test index. For a query $q$, a number of mappers and reducers (specialized to the query) are instantiated by the "Spawn" block in Fig. 4. The unit test index is read in parallel by these mappers. For a particular unit test of an API method, a mapper (that read it) evaluates the query (subexpression) on the test, to decide whether the API method matches the query. Every valid actuals-to-formals map between query variables and the method parameters, together with the outcome of the evaluation under it, is written out as a partial result. The distributed group-by operation then aggregates all partial results corresponding to the same API method. The reducer goes over these partial results – for each method, it picks the maximizing actuals-to-formals map, and assigns a score, using the maximizing test frequency. Subsequently, these methods are sorted based on score.

We now present a detailed explanation of the algorithms used in the mappers and reducers. Unit tests are read from a distributed index of unit tests as the mapper's input, as a stream of $\langle$key,value$\rangle$ pairs. The value $v$ in a pair $\langle k, v\rangle$ consists of two fields: $m$, that gives the signature of a method, and $\sigma$, that gives a mapping from the input/output variables to their values according to a unit test, as discussed in Section 3.1. In a MapReduce computation, a subset of such pairs is filtered for processing based on the key. Since we want to process every pair, we set the key $k$ to a fixed value, say null.

The mapper (see Algorithm 2) extracts the mapping $\sigma$ from the value, and constructs actuals-to-formals maps in a loop. It evaluates the query on a test input, under every actuals-to-formals map. The results of this evaluation are written as intermediate key-value pairs into the output stream by the function `Write`. An intermediate key-value pair $\langle m, (\lambda, true)\rangle$ or $\langle m, (\lambda, false)\rangle$, contains the method $m$ (as the key) and an actuals-to-formals map $\lambda$. This partial result says whether the query evaluated to true or false under a particular $\lambda \in \Lambda(q, m)$.

After the runtime performs a distributed group-by operation, a key-value pair arriving at a reducer (see Algorithm 3) contains a method $m$ (key) and a list of pairs (together constituting a value). Each pair $e$ has an actuals-to-formals map $e.\lambda$ and a Boolean value, $e.eval$, that was originally generated by evaluating the query using actuals-to-formals map $\lambda$ on a unit test of $m$. For each $\lambda$, we want to find the number of times the query evaluated to true under it, as well as the total number of unit tests

```
Input: Key-value pairs from the unit test
        index, Query $q \equiv x = e$
Output: A stream of intermediate keys
        indicating whether the query
        matches a unit test of a method,
        along with the actuals-to-formals
        map used
1 method MAP (key $k$, value $v$)
2     $m \leftarrow v.m$
3     $\sigma \leftarrow v.\sigma$
4     foreach $\lambda \in \Lambda(q, m)$ do
5         Construct $\sigma'$ from $\sigma, f$ and $\lambda$
6         if $\sigma'(x) = \sigma'(e)$ then
7             Write($\langle m, (\lambda, true) \rangle$)
8         else
9             Write($\langle m, (\lambda, false) \rangle$)
```

**ALGORITHM 2:** Mapper algorithm

```
Input: Output of the mappers grouped by
        method name
Output: The maximizing test frequency
        and the maximizing
        actuals-to-formals map for the
        input method
1 method REDUCE (key $k'$, value $list(v')$)
2     foreach $e \in list(v')$ do
3         $\lambda \leftarrow e.\lambda$
4         if $e.eval = true$ then
5             Count($\lambda$) $\leftarrow$ Count($\lambda$) $+ 1$
6         Total($\lambda$) $\leftarrow$ Total($\lambda$) $+ 1$
7     Let $\lambda^*$ be such that
      Count($\lambda^*$)/Total($\lambda^*$) is maximal
8     maxTestFreq $\leftarrow$ Count($\lambda^*$)/Total($\lambda^*$)
9     Write ($\langle k', (\lambda^*$,maxTestFreq$) \rangle$)
```

**ALGORITHM 3:** Reducer algorithm

it was evaluated on. So we iterate over the list, keeping track of these in Count($\lambda$) and Total($\lambda$) respectively (with both set to $0$ initially). Using these counters, we pick the maximizing actuals-to-formals map $\lambda^*$, and compute the maximizing test frequency for the method $m$. We then write this out as $\langle k', (\lambda^*,$ maxTestFreq$) \rangle$. The side-effect information for a method is computed a priori while building the distributed unit test index, and is stored as part of the method signature $m$.

*Example*. We illustrate the data-flow between mappers and reducers with an example. Suppose the mappers generate key-value pairs $\langle m_1, (\lambda_1, true) \rangle$, $\langle m_1, (\lambda_1, true) \rangle$, $\langle m_2, (\lambda_2, false) \rangle$, corresponding to methods $m_1$ and $m_2$ and the respective actuals-to-formals maps $\lambda_1$ and $\lambda_2$. We assume that we have two unit tests for $m_1$ which result in two intermediate key-value pairs, $\langle m_1, (\lambda_1, true) \rangle$, $\langle m_1, (\lambda_1, true) \rangle$. Similarly, we have only a single unit test for $m_2$. The input to the reducer(s) consists of $\langle m_1, [(\lambda_1, true), (\lambda_1, true)] \rangle$ and $\langle m_2, [(\lambda_2, false)] \rangle$. The output of the reducer consists of $\langle m_1, (\lambda_1, 1) \rangle$ and $\langle m_2, (\lambda_2, 0) \rangle$, indicating that $\lambda_1$ is the maximizing actuals-to-formals map for $m_1$ and $\lambda_2$, for $m_2$. The sort and print stage ranks $m_1$ with $\lambda_1$ as the most suitable API method and actuals-to-formals map for the query under consideration.

## 4. IMPLEMENTATION

*Unit Test Index Generation*. We instrumented the unit tests to record the values of parameters at the input and output, in addition to the value returned by the method (if any). We used `Serialysis`[9] to serialize these values to disk.

The input to our system is, for every method, a file containing the input/output data of its unit tests. We use a MapReduce job to process these files and build the unit test index. A mapper, with hooks for specifying the mapping between library and interpreter types, goes over these files and converts the input/output data from library data-types to interpreter data-types. This mapper is different from the mapper explained in Algorithm 2. It outputs a record containing key-value pairs that goes to the distributed unit test index (see Fig. 4).

*Unit Test Mining*. We implemented the MapReduce version of the mining algorithm (Algorithms 2-3) in the Apache `Hadoop` framework, with Scilab as the interpreter. Before invoking the algorithm, the search query is parsed, type-checked and decomposed

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

into subexpressions using the `Antlr3`[10] framework. A subexpression, together with key-value pairs read from the distributed unit test index, are inputs to the algorithm. The `Hadoop` implementation closely follows Algorithms 2-3, except that as an optimization, we query for every operator in the interpreted language offline, caching the top $k$ methods retrieved against each in an *operator index*. The operator index is a Java `HashMap`, serialized to disk.

*Snippet Generation*. When the user submits a query, we first decompose the query into subexpressions. We answer subexpressions in real-time by looking up the operator index, and then compose the results to emit pseudo-code snippets.

*Eclipse Plugin.* We implemented MATHFINDER as an Eclipse plugin that interfaces Eclipse with the query decomposition, unit test mining and snippet-generation engines. In Eclipse, the MATHFINDER view offers a search bar to type math expressions in, and displays the synthesized pseudo-code snippets library-wise. The user can copy-paste these snippets into the editor. The Eclipse plugin was used in our user study. The source code of the MATHFINDER Eclipse plugin, and the sequential and data-parallel mining algorithms, all the problems used in the user study, the unit-tests of the libraries indexed, the benchmark queries, and the migrated versions of Weka are publicly available at http://www.iisc-seal.net/mathfinder.

## 5. EXPERIMENTAL SETUP

We perform experiments to study the precision/recall of MATHFINDER on a large benchmark of queries, and evaluate the productivity gains obtained by using MATHFINDER in a user study. In addition, we study the performance and scalability of our technique and its effectiveness in API migration. This section describes our experimental setup.

### 5.1. Dataset Generation

*Target Libraries*. The target libraries we used were Colt, EJML, Jama and JBlas. Our retrieval target collection had 406 methods: 41 methods from Colt, 70 from EJML, 45 from Jama, and 250 from JBlas.

*Type Mappings*. For the target libraries mentioned above, we implemented the mapping between the library data-types and the interpreter data-types. Example snippets mapping the `Matrix` type of Jama, and the `DoubleMatrix` type of JBlas to the Scilab double matrix type are as follows:

Snippet mapping Jama's `Matrix`:

```
double[][] matrix = new double[][]{{1, 2}, {3, 4}};
Matrix m = new Matrix(matrix);
ScilabDouble d = new ScilabDouble(m.getArray());
```

Snippet mapping JBlas's `DoubleMatrix`:

```
double[][] matrix = new double[][]{{1, 2}, {3, 4}};
DoubleMatrix m = new DoubleMatrix(matrix);
ScilabDouble d = new ScilabDouble(m.toArray2());
```

The example above is illustrative. The code in our implementation differs from it by performing the type mapping in two stages. First, it converts the API types to a canonical intermediate representation to decouple them from the actual interpreter used. Then, it populates interpreter data-types from the intermediate representation. The

[10]antlr.org

code to perform the type mapping was less than 100 lines of Java for each of our target libraries. Mostly, the mapping was straight-forward; the biggest variation was that some libraries represented matrices as 2D arrays internally, or as 1D arrays in column-major order, where as our intermediate representation required a 1D row-major representation. Our experience suggests that other math libraries can easily be included as search targets. However, additional effort may have to be expended in writing code to perform the type mapping for domains where types are not closely aligned.

*Numerical Precision*. Recall that matching an API method to a subexpression involves comparing the result of evaluating the subexpression (using an interpreter) with the return value of the method. In our experiments, double precision numbers computed by the interpreter and by a method were considered equal if they were within $\epsilon = .1$ of each other. We allowed this relaxation to account for differences in the numerical precision of floating point computations performed by the interpreter and the third-party libraries. The relaxed matching results in the retrieval of spurious methods only in a few cases (mentioned in Section 6.1).

*Unit Testing*. The number and design of unit tests supplied as an input to our technique could potentially influence the experimental results. We did not use the test suites of our target libraries as input because the number of tests available per method was insufficient to study the relation between precision and the number of unit tests. Further, the test suites of Colt and JBlas did not exercise many methods.

We therefore wrote unit tests for the methods in the retrieval target collection using JUnit. Every unit test exercised a single API method. Hand-crafting unit test input values for a large number of tests is not only difficult, but could also introduce investigator bias. For this reason, we chose the input values used in the tests at random from a subset of values that respected the precondition of the method. For example, we ensured that the matrices supplied to matrix multiplication methods were of compatible dimensions. All the double matrices used as inputs to API methods had dimensions between 3 and 7 with their elements drawn uniformly at random between -1000 and 1000. We used multiple tests for each API method – in the evaluation of MATHFINDER (Sections 6 and 7), we report the results with 10 tests/method. We provide details about the relation between precision and the number of unit tests used in Section 8.

## 5.2. Validation of Results

In information retrieval, precision and recall are standard measures to evaluate the effectiveness of retrieval [Manning et al. 2008]. Precision refers to the fraction of retrieved results that are relevant, while recall is the fraction of relevant results that are retrieved. Both precision and recall therefore depend on classifying results based on relevance. In our setting, we had to ascertain whether API methods retrieved in the experiments to study MATHFINDER's precision (using queries from the user tasks, as well as the benchmark suite of queries) could be considered as relevant towards implementing the query. We also had to verify the correctness of the mined actuals-to-formals mappings and method composition.

To classify API methods as relevant and determine correct calling conventions for them, we analyzed the source code and documentation of the libraries in detail. We also looked at many API usage examples. In the case of methods that possessed insufficient documentation, or whose source code was hard to understand, we wrote unit tests to verify our understanding. To complement our manual validation of the pseudo-code snippets that were used to migrate Weka, we ran Weka's extensive regression suite on the migrated versions of the libraries (Section 7.2). Only one test failed in the case of each of the migrated versions.

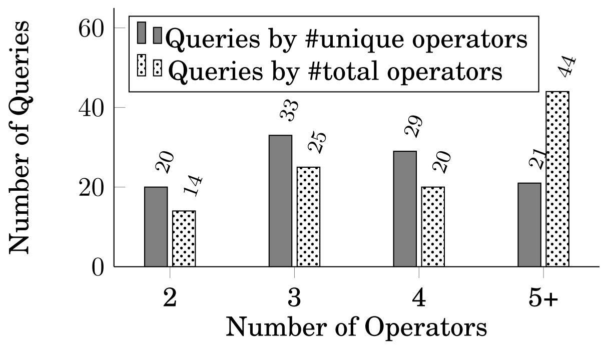

Fig. 5: Summary of the query collection to evaluate precision of pseudo-code snippet synthesis: The X-axis refers to the number of unique/total operators per query. The Y-axis is the number of queries.

## 6. EVALUATION ON API DISCOVERY TASKS

To evaluate our approach on API discovery tasks, we conduct an empirical evaluation using a benchmark query suite and a user study.

### 6.1. Empirical Evaluation over a Query Benchmark

In this section, we evaluate the precision and recall of MATHFINDER on a large benchmark suite of queries. These queries were culled from real-world algorithms drawn from diverse sources, and are indicative of API discovery tasks programmers encounter in practice. We describe the sources below.

*Sources*. While it is easy to evaluate precision using random queries generated from a grammar, this form of evaluation is not considered good since the queries will not reflect actual distribution of user requirements [Manning et al. 2008]. Therefore, we set up a benchmark by extracting expressions from Matlab programs accompanying literature on a diverse set of mathematical and engineering disciplines. It included the SOM Toolbox [Vesanto et al. 2000] that contains algorithms for clustering, classification, learning, and visualization; books on control systems [Datta 2004] and structural dynamics and civil engineering [Yuen 2010]; a numerical methods course [Persson 2007]; the BLAS specification [Dongarra 2002]; and a document on efficient Matlab usage [Acklam 2003]. Since we use Scilab as the math language interpreter in our implementation, we translated these expressions to Scilab.

*Query Collection*. We identified 115 unique operators used in these documents. We also culled 103 queries that would require method composition to implement, by manually identifying unique computations that involved composition of operators, and used them to evaluate precision of pseudo-code snippet synthesis. See Fig. 5 for the distribution of the query collection by the number of unique/total operators per query. The figure illustrates the overall sizes of the queries and their extensive use of different math operators. There were 9 queries with more than 10 operators each.

*Example*. The queries can be considered challenging inputs for MATHFINDER both because of the diversity of the operators in them, as well as the large number of operators in individual queries. To generate correct pseudo-code snippets, the methods corresponding to every operator have to be retrieved with high precision, and the retrieved methods have to be composed correctly, taking into account possible side-effects. For example, consider the following query from the k-means clustering algorithm of the SOM Toolbox [Vesanto et al. 2000] (with the type declarations we added):

```
double M data, centroids; int k, l; double dummyclusters;
dummyclusters = min(((ones(k, 1) ∗ sum((data.^2)', 1))' + ones(l, 1)
                ∗ sum((centroids.^2)',1) − 2.∗(data∗(centroids')))');
```

This expression uses 18 operators, of which 10 are unique, and makes extensive use of method composition. When it is used as a query (with type declarations added), MATHFINDER synthesizes the following pseudo-code from JBlas to implement it:

```
T0 = org.jblas.DoubleMatrix.ones(k, 1);
T1 = org.jblas.MatrixFunctions.pow(data, 2);
T2 = T1.transpose(); T3 = T2.columnSums(); T4 = T0.mmul(T3);
T5 = T4.transpose(); T6 = org.jblas.DoubleMatrix.ones(l, 1);
T7 = org.jblas.MatrixFunctions.pow(centroids, 2.);
T8 = T7.transpose(); T9 = T8.columnSums(); T10 = T6.mmul(T9);
T11 = T10.add(T5); T12 = centroids.transpose(); T13 = data.mmul(T12);
T14 = T13.mul(2); T15 = T11.sub(T14); T16 = T15.transpose( );
dummy = T16.min( );
```

The pseudo-code snippet shown is the direct output of MATHFINDER. It exercises 10 API methods, including static methods, that span two classes. The methods corresponding to all the distinct operators were correctly retrieved at rank 1, and our manual validation of the snippet established that MATHFINDER composed the methods correctly. This example illustrates that MATHFINDER can successfully synthesize API-driven pseudo-code snippets even for challenging queries.

*Results*. We now report the result of our precision study on this benchmark of queries. We evaluate the precision of our approach in two ways (1) whether it retrieves a correct method with a correct actuals-to-formals map for every operator queried, i.e., whether API discovery is precise, and (2) whether the pseudo-code snippets generated for queries (involving multiple operators) are correct. Some operators used in these queries were not supported by every library. An operator was considered unsupported by a library if it did not fetch results from the index, and the index did not have any relevant method from that library to implement it (as determined by exhaustive manual analysis of methods and their documentation). We only consider the operators supported by each library while calculating the precision, except when unsupported operators fetch results with scores above a threshold ($0.75$). The latter results are considered spurious, and we report precision having accounted for them.

The precision of API discovery was high (91% on average, as given in Table III), despite the fact that these libraries use different class definitions, calling conventions, etc. The study revealed a cause of imprecision – the operators ceil, floor and round that deal with converting double precision numbers to integers fetched spurious copy and clone methods from the libraries owing to the relaxed equality between query and method outcomes (see discussion on numerical precision in Section 5.1). Where recall is concerned, if a relevant method is included in our index, then it is always retrieved. The only exception proved to be the method `eyes` from JBlas, that implements the eyes operator for generating identity matrices. 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. We note that some libraries provide factory methods and constructors to

| Library | #correct@rank-1 / #supported-operators |
|---|---|
| Colt | 24/ 28(86%) |
| EJML | 36/ 37(97%) |
| Jama | 26/ 30(87%) |
| JBlas | 53/ 57(93%) |
| Total | 139/152(91%) |

Table III: Precision of API Discovery

| Library | #correct-snippet / #expressions |
|---|---|
| Colt | 12/ 12(100%) |
| EJML | 52/ 52(100%) |
| Jama | 39/ 39(100%) |
| JBlas | 68/ 72( 94%) |
| Total | 171/175( 98%) |

Table IV: Pseudo-code snippet synthesis precision

implement some operators, such as eye and zeros. We did not include factory methods and constructors in our index, and consequently, these were not retrieved.

To evaluate the precision on pseudo-code snippet synthesis, we used the 103 queries that would require method composition to implement. To measure the precision on a library, we only considered queries that could be implemented fully using it, i.e., every operator in the query was supported by the library. With this restriction, Colt supported 12 queries, EJML 52, Jama 39, and JBlas 72. For the expressions that could be implemented, we evaluated, for each library, whether the top-most code snippet MATHFINDER returned was correct. The precision of the synthesized pseudo-code snippets was 98% on average (as given in Table IV).

We note that MATHFINDER mines only unit tests of individual methods, and not tests of compositions of API methods. The precision on the pseudo-code snippet synthesis, in spite of this restriction, constitutes evidence that MATHFINDER can synthesize correct pseudo-code snippets. 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.

## 6.2. 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.

*6.2.1. Users and Tasks.* We picked a set of four mathematical programming tasks (see Table V for a summary) that required third-party libraries to complete. We selected these tasks for the following reasons. First, they are general tasks – well known algorithms drawn from linear algebra, machine learning and control theory. Second, we deliberately chose small tasks, expecting the participants to finish them within two hours. The main barrier to implementation was the lack of direct Java support, rather than algorithmic subtleties. There were 16 unique operators across the tasks, and 5-8 queries in each task whose implementation required method composition. We presented the math specification of the algorithms to the participants as comments inside a Java method stub. Only the algorithm for conjugate gradient computation could be implemented using any one of the target libraries, while the others required a careful evaluation of the APIs to find a functionally complete target library. We ensured that for every task, at least one of the four libraries chosen was functionally complete.

We deemed participants to have completed a task when their program passed all our unit tests – these tests were for the algorithm the programmers implemented and were independent of the unit tests used by MATHFINDER. Our participants were 5 industry professionals and 3 graduate students not affiliated with our research group. None of the participants had prior exposure to the target libraries or interpreted lan-

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

Table V: Tasks used in the user study

| 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) |

Table VI: Task completion times

guages like Matlab or Scilab. Out of the participants, 4 had more than 2 years of experience using Java, 2 had a year's experience, while the remaining had more than 4 years of experience. Every task was attempted by a pair of participants with similar levels of Java expertise. One of each pair, picked at random, was allowed to use MATHFINDER. This corresponds to a *matched-pairs* block design with explicit control for the level of experience as a potential confounding variable. Those in the *control group* were allowed to use Javadoc, Eclipse code completion, and web or code search engines (including Codase, Google, Koders and Krugle); in addition, those in the *experimental group* were allowed to use MATHFINDER. We gave the experimental group participants a 20 minute presentation on the tool before the study. The names of the algorithms shown in Table V were not revealed to the participants lest they were to find a complete implementation on the web. We gave all the participants handouts describing the operators used in the tasks, and a mapping between library types and math types (similar to Table I).

We did not provide type-based API discovery tools [Mandelin et al. 2005; Thummalapenta and Xie 2007] to aid participants, since we believed that these tools 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.

*6.2.2. Results.* To evaluate the effectiveness of the MATHFINDER approach, we investigated the following research questions:

*RQ1*: *Productivity* - Does MATHFINDER improve programmer productivity?
*RQ2*: *Precision and Recall* - Are the results returned by MATHFINDER precise? Does MATHFINDER return all relevant results (recall)?

*(RQ1) Productivity*. Table VI 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. The control group users took 90 minutes to complete the task on average, whereas the experimental group users took 46 minutes on average. We also evaluate the statistical significance of the results using a *matched pairs t-test*, which is a method to test hypotheses about the means of two matched groups to detect whether there are any statistically significant differences between these means. The null hypothesis in our case states that there is no difference between the mean time taken by the control group users and the experimental group users, and the observed difference is due to randomness. The two tailed t-statistic for the matched pairs block design was significant at the $.05$ critical $\alpha$ level with $t(3) = 7.42$ and $p = .005$. The $p$ value is the probability of obtaining the observed results, assuming that the null hypothesis is true. Given that the $p$ value is less than our significance level $\alpha$, we reject the null hypothesis and conclude that MATHFINDER helps improve programmer productivity.

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

Table VII: Precision of API Discovery

| Library | #correct-snippet / #expressions |
|---|---|
| Colt | 2/ 6( 33%) |
| EJML | 17/17(100%) |
| Jama | 15/15(100%) |
| JBlas | 16/16(100%) |
| Total | 50/54( 93%) |

Table VIII: Precision of synthesized pseudo-code snippets

We observed that MATHFINDER users 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 in almost all cases, 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. On the other hand, while query formulation did not consume time for control group participants (they formulated keyword-based queries using the handout we provided them), they had to sift through results returned by search engines and read the Javadoc for multiple libraries to discover relevant methods, and then learn how to use them. This experience leads us to believe that MATHFINDER will deliver larger productivity gains with more complex tasks and diverse API requirements.

*(RQ2) Precision and Recall.* To evaluate precision of API discovery, we picked the set of unique operators from across the tasks used in the user study; 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 VII. The precision is $96\%$ on average. Also, MATHFINDER retrieved all relevant methods from all libraries (recall 1), with two exceptions. One was the eyes operator, used to generate identity matrices (owing to the mismatch in the number of arguments between an operator and an equivalent method, as explained in Section 6.1). The other exception was the transpose operator. MATHFINDER mapped it to an incorrect method of Colt. Later, we were able to attribute it to having missed a special case in mapping Colt's `DenseDoubleMatrix` type to the interpreter data-type. This implementation issue was easy to fix, but we only report results prior to the fix in Table VII.

There were 24 queries in total across all the tasks. As before, to measure the precision of pseudo-code snippet synthesis on a library, we only considered expressions that could be implemented fully using it. The results of the precision of pseudo-code snippet synthesis are given in Table VIII. The precision across libraries is $93\%$ on average. Colt's precision was low because 4/6 expressions used transpose (which was mapped to an incorrect method, as discussed earlier).

### 6.3. Threats to Validity

The empirical evaluation of the precision and recall of MATHFINDER used a benchmark suite of queries we chose, introducing the threat of investigator bias. To mitigate this threat, the benchmark included every operator from our sources that could be expressed in the query language (i.e., was supported by Scilab), resulting in 115 distinct operators. When picking queries for pseudo-code snippet synthesis, we only discarded those queries whose operators were already exercised by some other query already in the benchmark, i.e., expressions in programs that constituted our sources were not discarded in a biased fashion. A threat to *external validity* is that the results may not generalize to queries drawn from other sources. To mitigate this threat, we made every effort to ensure that the queries were representative of actual user requirements by drawing them from diverse sources.

The user study may suffer from some 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 used a matched pairs block design that explicitly controlled for mismatched levels of experience, since it is likely that experienced users could finish their tasks faster. In the study, programmers with similar levels of Java expertise were given the same task, but their group was chosen randomly. To mitigate testing bias, we avoided giving the experimental group participants a hands-on tutorial, that might have familiarized them with the tool, and gave 20 minute presentation instead. 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.

## 7. EVALUATION ON AN API MIGRATION CASE STUDY

To assess whether MATHFINDER can aid the migration of software from a third-party math library to potential target libraries, we carried out the migration of Weka [Hall et al. 2009], a popular machine learning framework. Weka uses Jama for matrix and linear algebra computations. However, Jama is a reference implementation that has not seen active development since 2005. A benchmarking study [Abeles 2010] notes that there are newer and faster math libraries available. We picked two of these, EJML and JBlas as targets for our migration, in addition to Colt, an older library. We also migrated a deprecated Matrix class that the developers of Weka themselves wrote (`weka.core.Matrix`). We used Weka 3.6.5[11] in our study.

### 7.1. Methodology

Potential ways to migrate an application include replacing the source API by a wrapper based re-implementation that uses the target API [Bartolomei et al. 2010a], or inlining calls to methods of the source API by replacement code from the target API [Perkins 2005]. We didn't follow the former approach because of the potential performance penalty incurred in the wrappers (for object construction and data-type conversion), and the latter because of the large number of calls to methods of the source API in our case study. Instead, our approach was to identify all statements that referred to types from the source API and replace such statements by equivalent statements that used the target API instead.

The case study spanned all the classes of Weka that made use of Jama. There were 20 such classes in all, with 116 methods that used the source library. In all, there were 432 statements that referred to Jama or Weka's deprecated `Matrix` class. We identified

[11] http://sourceforge.net/projects/weka/files/weka-3-6/3.6.5/

| Class | #Sites | #Method Calls | #Constructors/ Declarations | #Field Accesses/ Other |
|---|---|---|---|---|
| LatentSemanticAnalysis | 32 | 23 | 8 | 1 |
| PrincipalComponents | 2 | 1 | 1 | 0 |
| PrecomputedKernelMatrixKernel | 3 | 0 | 3 | 0 |
| PaceMatrix | 85 | 13 | 8 | 64 |
| ChisqMixture | 7 | 7 | 0 | 0 |
| MixtureDistribution | 5 | 5 | 0 | 0 |
| NormalMixture | 4 | 2 | 0 | 2 |
| GaussianProcesses | 27 | 18 | 8 | 1 |
| PaceRegression | 5 | 5 | 0 | 0 |
| BFTree | 5 | 2 | 2 | 1 |
| SimpleCart | 5 | 2 | 2 | 1 |
| sIB | 43 | 25 | 13 | 5 |
| Optimization | 22 | 15 | 4 | 3 |
| LinearRegression | 20 | 15 | 5 | 0 |
| MahalanobisEstimator | 26 | 20 | 5 | 1 |
| NNConditionalEstimator | 7 | 4 | 2 | 1 |
| PLSFilter | 107 | 51 | 55 | 1 |
| XMLBasicSerialization | 16 | 2 | 12 | 2 |
| ConfusionMatrix | 10 | 9 | 1 | 0 |
| CostMatrix | 1 | 0 | 0 | 1 |
| **Total**: 20 | 432 | 219(51%) | 129(30%) | 84(19%) |
| **Approach Used**: |  | MATHFINDER | Semi-automated | Manual |

Table IX: Classification of Migration Sites

these statements using a small program written using the Eclipse JDT framework[12]. For the migration of statements involving calls to Jama's methods, we framed math queries and used MATHFINDER to obtain equivalent pseudo-code snippets from Colt, EJML and JBlas. There were 219 such statements and their migration could be carried out using our tool (column "#Method Calls" in Table IX).

Other statements that had to be migrated included field and method declarations, and constructor invocations. There were 129 such statements. We used a rewrite-rule based approach [Nita and Notkin 2010] to migrate such statements to target libraries. The rewrite rules mapped Jama types to types in the target libraries, and could be supplied manually as we had to migrate only 4 Jama types. This set of statements is shown under the column "#Constructors/Declarations" in Table IX. Their migration was performed in a semi-automated manner.

We migrated the remaining 84 statements manually. These were mainly from the `PaceMatrix` class of Weka that inherited Jama's `Matrix` class. Many statements in the `PaceMatrix` class directly accessed a data-structure field of the `Matrix` class, and had to be translated to work with data-structures that corresponding target library types used. For example, during migration to JBlas, a 2D array access had to be translated to accesses into a column-major 1D array. To the best of our knowledge, there is no automatic technique to migrate such data-structure accesses. In addition, the semantics of methods like `copy` and `clone` as well as methods that wrote and read from files were not expressible in our query language - we migrated calls to such methods manually.

A fine-grained class-wise view of our classification of migration sites is presented in Table IX. Overall, 219 statements were considered for querying MATHFINDER.

[12]www.eclipse.org/jdt/

| Class | #Method Calls | EJML | JBlas | Colt |
|---|---|---|---|---|
| LatentSemanticAnalysis | 23 | 19/23(83%) | 19/23(83%) | 19/23(83%) |
| PrincipalComponents | 1 | 0/1(0%) | 0/1(0%) | 0/1(0%) |
| PrecomputedKernelMatrixKernel | 0 | - | - | - |
| PaceMatrix | 13 | 13/13(100%) | 13/13(100%) | 12/12(100%) |
| ChisqMixture | 7 | 7/7(100%) | 7/7(100%) | 7/7(100%) |
| MixtureDistribution | 5 | 5/5(100%) | 5/5(100%) | 5/5(100%) |
| NormalMixture | 2 | 2/2(100%) | 2/2(100%) | 2/2(100%) |
| GaussianProcesses | 18 | 18/18(100%) | 17/18(94%) | 17/18(94%) |
| PaceRegression | 5 | 5/5(100%) | 5/5(100%) | 5/5(100%) |
| BFTree | 2 | 0/2(0%) | 0/2(0%) | 0/2(0%) |
| SimpleCart | 2 | 0/2(0%) | 0/2(0%) | 0/2(0%) |
| sIB | 25 | 25/25(100%) | 25/25(100%) | 25/25(100%) |
| Optimization | 15 | 15/15(100%) | 15/15(100%) | 15/15(100%) |
| LinearRegression | 15 | 15/15(100%) | 15/15(100%) | 14/14(100%) |
| MahalanobisEstimator | 20 | 20/20(100%) | 20/20(100%) | 20/20(100%) |
| NNConditionalEstimator | 4 | 4/4(100%) | 4/4(100%) | 4/4(100%) |
| PLSFilter | 51 | 49/51(96%) | 48/51(94%) | 48/51(94%) |
| XMLBasicSerialization | 2 | 2/2(100%) | 2/2(100%) | 2/2(100%) |
| ConfusionMatrix | 9 | 9/9(100%) | 9/9(100%) | 9/9(100%) |
| CostMatrix | 0 | - | - | - |
| **#correct-snippet@rank-1/#queries** | 219 | 208/219(95%) | 206/219(94%) | 204/217(94%) |

Table X: Precision on API migration.

## 7.2. Results

*Effectiveness of API Migration.* We report the precision of API migration in terms of the number of statements to be migrated using MATHFINDER (listed under the column "#Method Calls" in Table X). In case a statement has both API elements that are amenable to migration using our tool (method calls), and those that are not, we count it under "#Method Calls". If a statement has multiple API methods to be migrated, and a correct result is not retrieved in the top most position for even one, we do not count it as correctly retrieved. As before, we exclude operators not implemented by a library when studying precision on it (which is why only 217 sites appear under the Colt library).

In all, there were 17 unique operators that had to be migrated across the 20 classes. We report precision on the pseudo-code snippets obtained in Table X. The columns labeled with library names indicate the number of statements for which MATHFINDER returned a correct pseudo-code snippet from those libraries (of those statements under the purview of the tool). The precision observed for migration to EJML is $95\%$ and that for Colt and JBlas are $94\%$. The average precision is $94\%$.

Relevant API methods were not retrieved in the case of math operators that could be implemented in different ways, each yielding a distinct but correct result. An example for such an operator is eigenvector computation – an eigenvector $x$ of a matrix $A$ is any vector such that $Ax = \lambda x$ for some scalar eigenvalue, $\lambda$. In general, there may be an infinite number of (linearly dependent) eigenvectors corresponding to each eigenvalue, and any one of these can be considered correct. The eigenvectors computed for a matrix by the interpreter and the libraries, were, in general, different. The order of the eigenvalues returned by the interpreter also differed from the order in which the API methods returned them. Consequently, methods for obtaining eigenvectors and eigenvalues following the eigenvalue decomposition of a matrix were not retrieved from any of the libraries. The methods pertaining to the singular value decomposition (svd) of a matrix were also not retrieved for similar reasons. For these reasons, we used wrapper based implementations to migrate these methods.

The high precision obtained across multiple libraries in a realistic case study, and the large number of statements under the purview of MATHFINDER together support the claim that MATHFINDER is a valuable tool to aid the migration of clients using math libraries. In addition, MATHFINDER helped address another challenging aspect of API discovery and migration – many real world APIs place methods pertaining to a "main" object in other, helper objects. Unfortunately, programmers are slower in finding other related objects that are not referenced in the main object [Stylos and Myers 2008]. For example, `DoubleMatrix` is the main matrix class in JBlas, but many methods operating on double matrices are in a helper class `MatrixFunctions`, and none of the method signatures of methods in `DoubleMatrix` refer to `MatrixFunctions`. In fact, all the libraries we surveyed had methods pertaining to linear algebra and matrix computations spread across multiple classes. This necessitates a many-to-many migration, where functionality in source classes are supplied by multiple target classes. A class level summary of migrating Jama to EJML and JBlas which we derived from MATHFINDER 's results is shown in Fig. 6. As can be seen, the functionality of the Jama `Matrix` class is spread across multiple classes in both EJML and JBlas.

A class that proved challenging to migrate was the `PaceMatrix` class that inherited from the Jama `Matrix` class. Many methods of the `PaceMatrix` class directly manipulated the internal data structures used by the `Matrix` class and we had to port these operations manually to the data structures used by matrix classes of the target libraries. This was non-trivial – for example, JBlas maintains an invariant on its `DoubleMatrix` class that says that the length of the array it uses to store matrices is always equal to the product of its row and column fields. Many of `PaceMatrix`'s methods violated this invariant since Jama does not enforce it, and consequently, these methods proved difficult to migrate. To the best of our knowledge, such challenges place these sites beyond the state-of-the-art in automatic migration techniques. Additionally, migrating `PaceMatrix` also necessitated the migration of 4 other classes that used it. In some cases, none of the target libraries had functionality that was provided by the source library. For example, the `PrecomputedKernelMatrixKernel` class used a constructor of the `Matrix` class of Jama that allows a matrix to be populated from a file on disk. Although there were constructors in the target library classes that could populate matrices from files, none of them understood the specific format that Jama expected.

Overall, our evaluation indicates that in a large number of cases, MATHFINDER provided pseudo-code snippets that could be used directly during migration. Since we performed the migration over several days, it is difficult to estimate the exact gain in programmer productivity. Nevertheless, the extent of our case study is large (as seen in Table IX and Table X), and the three target libraries we migrated to had diverse class hierarchies and coding conventions. MATHFINDER produced uniformly precise snippets across all three.

*Querying for Reuse Opportunities*. We observed that migration can lead to additional opportunities for reuse. The developers of Weka wrote methods to perform some math computations that Jama does not support. Some of these were supported by the target libraries. Retaining such client methods after migration leads to suboptimal use of the target library's API – the client could call the method from the target library instead of reimplementing it. For example, the `maxAbs` method in the `PaceMatrix` class of Weka returns the maximum absolute value of all elements in a matrix. We can issue the query shown below to search for an implementation in the target libraries:

*double M* v; *double* r; r = max(abs(v));

MATHFINDER suggests this snippet from JBlas

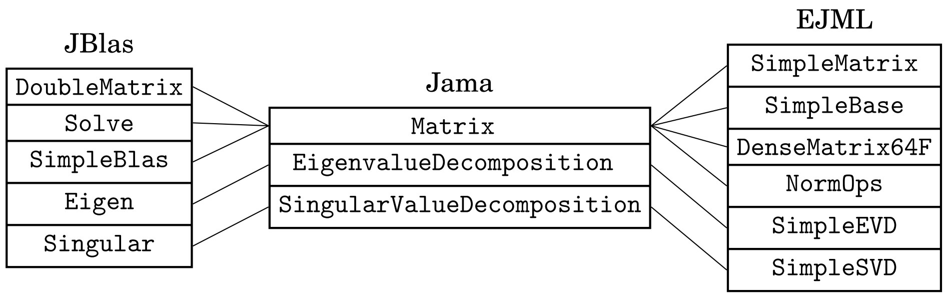

Fig. 6: Class level summaries for migrating from Jama to JBlas and EJML. An edge between $c$ and $c'$ indicates that there exists a method in $c$ for which an equivalent implementation may be found in $c'$

```
DoubleMatrix v, T1; double r; T1 = DoubleFunctions.abs(v); r = T1.max();
```

If the programmer chooses to search at the granularity of the entire expression, then MATHFINDER recommends the following snippet from JBlas.

```
DoubleMatrix v; double r; r = v.normmax()
```

The programmer will then have to either wrap the appropriate implementation from JBlas in `maxAbs` or refactor the client call sites that use `maxAbs` appropriately.

*Validation of Migration Results.* In addition to manually validating the snippets returned by the tool using our knowledge of the target libraries, we ran Weka's extensive regression suite on the migrated versions of the libraries. The suite consisted of 5305 tests in all. We ran the regression tests on all the migrated versions. 5304 tests passed in each case.

Subsequent analysis of the regression suite revealed that methods from 17 of the 20 classes involved in migration (with the exception of `MahalanobisEstimator`, `NNConditionalEstimator` and `XMLBasicSerialization`) were exercised in 356 tests from the suite. The analysis also helped pinpoint the causes for the failing tests. In the case of the code migrated to Colt, a test failed because of a numerical accuracy issue - the expected and actual values differed in the $14^{th}$ digit following the decimal point. In the versions of Weka migrated to EJML and JBlas, a test failed because while the matrix multiplication method of Jama can accept matrices with one of the dimensions being 0, the corresponding methods in the target libraries throw exceptions in this case.

The success of the migrated versions of Weka on Weka's large regression test suite gives strong evidence that MATHFINDER can provide an accurate mapping between APIs to support migration.

## 7.3. Threats to Validity

There is no "ground truth" to evaluate the precision of the pseudo-code snippets mined by our tool, and therefore, the correctness of migration. We mitigate this threat by carefully studying the documentation and test cases of the target libraries to classify the retrieved snippets as correct. Weka's regression tests pass on all the migrated versions, providing further evidence for the correctness of migration. We also note that we framed the queries ourselves. This is not a threat to the reported precision and recall, since they have been independently established as high by the empirical evaluation on the query benchmark (see Section 6.1). However, if queries do not accurately reflect the semantics of the statements to be migrated, while correct results to the query might to be retrieved, the retrieved pseudo-code snippet would reflect the semantics of the query rather than that of the statement to be migrated. Further studies are required to address the question of whether users will be able to frame queries that accurately reflect the semantics of the statements to be migrated. Since it is reasonable to assume that users who undertake the migration task understand the client source code, they should be able to express the statement semantics in terms of mathematical operations correctly.

## 8. DISCUSSION

In this section, we study the scalability of our technique as the number of tests available for each API increases, and the relation between precision of MATHFINDER 's results and the number of test records. We also discuss the limitations of our work and possible extensions to our approach.

*Scalability*. We obtain the time for API discovery using 10, 200 and 500 tests/method against queries involving operators used in the tasks listed in Table V. 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). Recall that our index comprised over 400 methods across the four libraries (see Section 5.1). With 500 tests/method, we obtain an index with over 200K tests. MATHFINDER mines suitable APIs over this index in 80.5s on average, demonstrating that the MapReduce algorithm (Algorithm 2) scales well to large data sizes.

*Precision vs. Unit Tests*. The precision of results reported in Sections 6 and 7 is with respect to 10 tests/method. The precision did not vary 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. Given the scalability of MATHFINDER, it is therefore fair to assume that it can support retrieval over a large number of libraries.

*Limitations*. Our ranking function does not take into account performance or efficiency of API implementations. Some API methods implement specializations of math operators, i.e., when the operator is queried for, these methods are not ranked lower than their general counterparts. For example, a query for the operator pinv to compute the pseudoinverse of a matrix will retrieve methods for computing the inverse as well. This is because the pseudoinverse and inverse coincide for invertible matrices, and the tests of the inverse methods only took invertible matrices as their input. We cannot discover methods that are non-deterministic, and the semantics of some methods are not expressible in our query language (e.g. `copy/clone`). We cannot discover compositions of API methods to implement a single operator. Our approach, in its current form, cannot discover higher-order 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. Spurious results may be obtained as a result of relaxing the equality between query and method outcomes. Further, the incompleteness or errors in data (unit tests) can affect precision of results. This is true of any data mining approach. We plan to address some of these limitations in the future.

*Future Work*. 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 over math variables. We believe that our approach of mining unit tests can be useful in domains other than math APIs. We are currently exploring this possibility.

## 9. RELATED WORK

### 9.1. API Discovery

*Syntactic Approaches*. 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.)[13] 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 [Hoffmann et al. 2007], Codifier [Begel 2007], Sourcerer [Linstead et al. 2009] and Portfolio [McMillan et al. 2011] can perform syntactic search using richer program structure. The MATHFINDER approach is purely *semantic* and does not use keywords or program structure for search.

*Type Based Approaches.* Several approaches [Rittri 1990; Zaremski and Wing 1993; Mandelin et al. 2005; Thummalapenta and Xie 2007] 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 [Yessenov et al. 2011], 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 [Ye and Fischer 2002; Holmes and Murphy 2005; Sahavechaphan and Claypool 2006; Duala-Ekoko and Robillard 2011]. Types are also combined with keywords in Keyword Programming [Little and Miller 2007] and SNIFF [Chatterjee et al. 2009]. Apart from APIs, type-based code completion approaches such as InSynth [Gvero et al. 2011] and the work of Perelman et al. [Perelman et al. 2012] also search over variables in the typing context.

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.

*Other Approaches*. Prime [Mishne et al. 2012] 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* [Zaremski and Wing 1997; Reiss 2009] and *test-*

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

*driven* [Podgurski and Pierce 1992; Hall 1993; Hummel et al. 2008; Reiss 2009; Lemos et al. 2011] techniques for API discovery.

### 9.2. API Migration

Teyton et al. [Teyton et al. 2012] and Bartolomei et al. [Bartolomei et al. 2010b] point out that migration of a client from a library to another functionally equivalent library is a common software development activity owing to concerns about performance, obsolescence, security, compatibility, convenience, or licensing.

*Migration of Version-related APIs.* As libraries and frameworks evolve, clients that use them have to update their code to make use of the new versions, since not all changes are backwards compatible. Dig and Johnson [Dig and Johnson 2006] observe in their case studies that the majority of the changes to an API that break backward-compatibility are standard refactorings.

To help the automatic migration of clients to newer versions of the API, tools like CatchUp [Henkel and Diwan 2005] that record the refactorings made by the API developer have been proposed. The refactorings are then replayed to migrate clients. Since refactoring logs may not always be available, algorithms to infer refactorings between API versions have been proposed [Dig et al. 2006; Weissgerber and Diehl 2006].

However, migration may involve handling changes other than standard refactorings. Research efforts have targeted mining method correspondences, both one-to-one [Dagenais and Robillard 2009; Schäfer et al. 2008], one-to-many and many-to-one [Wu et al. 2010], as well as many-to-many [Meng et al. 2012]. Work by Schäfer et al. [Schäfer et al. 2008] mines rules that encode correspondences between classes and fields as well, while LibSync [Nguyen et al. 2010] recommends edit operations on client code that calls into the API, to adapt it to use the newer API. These edit operations go beyond changing the target of calls to use the newer version of the API to inserting the control structure that could potentially surround these calls. Kapur et al. [Kapur et al. 2010] address the problem of refactoring references that are rendered dangling as a result of library migration. For a detailed survey of the literature on the problem of version related migration, we refer the reader to [Robillard et al. 2013].

Tools to help migration to a newer version of an API by mining either API code or client code typically make use of similarities in the text or structure, heuristics based on metrics or call-dependency, graph based models of the API, or combinations of these to infer correspondences between methods (or API elements). For these techniques to be applicable at the call sites that MATHFINDER targets, pairs of migrated clients, one per target library would have to be found. Moreover, these approaches assume that the source and target libraries are version related and it is not clear as to how their precision will be affected when this assumption no longer holds.

While MATHFINDER can be seen as a tool to help migration by discovering one-to-one method correspondences between the source and target APIs, it addresses the setting where the source and target APIs can be arbitrarily different in their structure, and in the naming conventions that they follow. It supports queries involving method composition, and by changing the granularity of query decomposition, it can also find many-to-one mappings. Cossette and Walker [Cossette and Walker 2012] observe that sometimes, a library removes functionality it originally provided, so that client callers have to replace that functionality with equivalent methods from other libraries. They classify such a transformation as hard to automate. The math expressions we used to encode source method semantics and query for equivalent target methods can be used to guide this transformation.

*Rewrite-rule Based Migration.* Researchers have proposed techniques to address migration between non-version related libraries [Balaban et al. 2005; Nita and Notkin

2010]. These approaches take rewrite rules in the form of mappings between source API elements and their replacements in the target as input, and automate migration to a large extent, offering some guarantees on the type correctness of the migrated code. For the migration study that we undertook, we used rewrite rule based techniques to migrate constructors and declarations. The number of types to be migrated was relatively small, making it feasible for a developer to provide rewrite rules to migrate declarations and constructors as an input.

Techniques have been proposed to mine mappings between source and target API elements automatically; the MAM project [Zhong et al. 2010] uses Java applications that have been ported to C# to infer how a source API element maps to a target API element. The existence of code with the same functionality in the two languages allows the 'aligning' of classes and methods by name similarity. The inference results are refined via the analysis of API structure including fields, method parameters and local variables of methods. Rosetta [Gokhale et al. 2013] uses functionally equivalent client programs – one that uses the source API and the other the target API – to mine mappings. Rosetta runs a probabilistic inference algorithm on traces of API calls invoked by the two applications when they are executed in similar ways, to output a ranked list of target API methods that a particular source API method maps to. MATHFINDER uses a mathematical specification and the unit tests of multiple libraries to mine a set of methods that implement the specification. It does not assume the existence of pairs of similar client applications, one written in the source library, and the other in some target library.

Most sites that had to be migrated manually during our case study were accesses to superclass fields in the source library. While the work of Schäfer et al. [Schäfer et al. 2008] could potentially be used to map source fields to target fields, translating access of superclass fields (from the source library) into equivalent access idioms for fields from the target library (see Section 7.1 for an example) remains a challenging problem.

*Reuse and Detecting Reimplemented Methods*. Kawrykow and Robillard [Kawrykow and Robillard 2009a; Kawrykow and Robillard 2009b] study the problem of missed opportunities for API reuse where client code could re-implement the functionality of an API method instead of calling it. Their technique warns developers of potentially reimplemented API methods automatically. Although our technique is specific to math APIs and requires manual queries from the developer, we aim at detecting methods that become redundant on moving to a *different* API. We note that our technique can also be used to detect reimplementation of methods from the API a client currently uses. The Gilligan tool suite [Holmes and Walker 2013] addresses the more general problem of supporting software reuse tasks even when the code to be reused may not have been designed with such tasks in mind.

## 10. CONCLUSIONS

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 have applied this technique to both API discovery and migration for math APIs. We evaluated MATHFINDER 's precision on a large benchmark of queries. In this evaluation, the precision of API discovery was 91% and the precision of pseudo-code synthesis was 98%. We conducted a user study to evaluate the productivity gains obtained by using MATHFINDER for API discovery. The programmers who used MATHFINDER finished their programming tasks twice as fast as their counterparts who used the usual techniques like web and code search, IDE code completion, and manual inspection of library documentation. For the problem of API migration, as a case study, we used MATHFINDER to migrate a popular machine learning library, Weka, to matrix and linear algebra libraries that are more efficient and well-supported than the Jama library it uses currently. The migration task spanned 20 classes and 116 methods. In 94% of the queries, MATHFINDER returned correct pseudo-code snippets from the target APIs. Our implemenatation based on a data-parallel version of the unit test mining algorithm scaled to a large collection of unit tests consisting of over 200K tests and returned results in 80.5s on average, on an 8-core machine. In conclusion, we show that the approach improves programmer productivity, gives precise results, and scales to large data-sets.

**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. This work is partially supported by the Robert Bosch Center for Cyber Physical Systems, Indian Institute of Science.

## REFERENCES

P. Abeles. 2010. Java Matrix Benchmark. http://code.google.com/p/java-matrix-benchmark/. (2010). Accessed: 2012-07-30.

P. J. Acklam. 2003. MATLAB array manipulation tips and tricks. http://home.online.no/~pjacklam/matlab/doc/mtt/doc/mtt.pdf. (Oct. 2003).

A.V. Aho, M. Lam, R. Sethi, and J.D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools*. Prentice Hall.

I. Balaban, F. Tip, and R. Fuhrer. 2005. Refactoring Support for Class Library Migration. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 265–279.

T.T. Bartolomei, K. Czarnecki, and R. Lammel. 2010a. Swing to SWT and back: Patterns for API migration by wrapping. In *Proceedings of the International Conference on Software Maintenance*. 1 –10.

T. T. Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm. 2010b. Study of an API migration for two XML APIs. In *Proceedings of the International Conference on Software Language Engineering*. 42–61.

A. Begel. 2007. Codifier: a programmer-centric search user interface. In *Workshop on Human-Computer Interaction and Information Retrieval*.

S. Chatterjee, S. Juvekar, and K. Sen. 2009. SNIFF: A Search Engine for Java Using Free-Form Queries. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*. 385–400.

B. E. Cossette and R. J. Walker. 2012. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 55:1–55:11.

B. Dagenais and M. P. Robillard. 2009. SemDiff: Analysis and recommendation support for API evolution. In *Proceedings of the International Conference on Software Engineering*. 599–602.

B.N. Datta. 2004. *Numerical Methods for Linear Control Systems*. Elsevier Inc.

J. Dean and S. Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113.

D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. 2006. Automated detection of refactorings in evolving components. In *Proceedings of the European Conference on Object-Oriented Programming*. 404–428.

D. Dig and R. Johnson. 2006. How do APIs evolve? A story of refactoring. *J. Softw. Maint. Evol.* 18, 2 (March 2006), 83–107.

J. Dongarra. 2002. Basic Linear Algebra Subprograms Technical (Blast) Forum Standard (1). *IJHPCA* 16, 1 (2002), 1–111.

E. Duala-Ekoko and M. P. Robillard. 2011. Using structure-based recommendations to facilitate discoverability in APIs. In *Proceedings of the European Conference on Object-oriented programming*. 79–104.

A. Gokhale, V. Ganapathy, and Y. Padmanaban. 2013. Inferring likely mappings between APIs. In *Proceedings of the International Conference on Software Engineering*. 82–91.

T. Gvero, V. Kuncak, and R. Piskac. 2011. Interactive synthesis of code snippets. In *Proceedings of the International Conference on Computer Aided Verification*. 418–423.

M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. 2009. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 10–18.

R. J. Hall. 1993. Generalized behavior-based retrieval. In *Proceedings of the International Conference on Software Engineering*. 371–380.

J. Henkel and A. Diwan. 2005. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proceedings of the International Conference on Software Engineering*. 274–283.

R. Hoffmann, J. Fogarty, and D. S. Weld. 2007. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the ACM Symposium on User Interface Software and Technology*. 13–22.

R. Holmes and G. C. Murphy. 2005. Using structural context to recommend source code examples. In *Proceedings of the International Conference on Software Engineering*. 117–125.

R. Holmes and R. J. Walker. 2013. Systematizing Pragmatic Software Reuse. *ACM Trans. Softw. Eng. Methodol.* 21, 4 (Feb. 2013), 20:1–20:44.

O. Hummel, W. Janjic, and C. Atkinson. 2008. Code Conjurer: Pulling Reusable Software out of Thin Air. *IEEE Softw.* 25, 5 (Sept. 2008), 45–52.

P. Kapur, B. Cossette, and R. J. Walker. 2010. Refactoring references for library migration. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. 726–738.

D. Kawrykow and M.P. Robillard. 2009a. Detecting inefficient API usage. In *Proceedings of the International Conference on Software Engineering, Companion*. 183 –186.

D. Kawrykow and M. P. Robillard. 2009b. Improving API Usage through Automatic Detection of Redundant Code. In *Proceedings of the International Conference on Automated Software Engineering*. 111–122.

G. T. Leavens, A. L. Baker, and C. Ruby. 1999. JML: A Notation for Detailed Design. In *Behavioral Specifications of Businesses and Systems*. 175–188.

O. A. L. Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. 2011. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Inf. Softw. Technol.* 53, 4 (April 2011), 294–306.

E. Linstead, S. K. Bajracharya, T. C. Ngo, P. Rigor, C. V. Lopes, and P. Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.* 18, 2 (2009), 300–336.

G. Little and R. C. Miller. 2007. Keyword programming in Java. In *Proceedings of the International Conference on Automated Software Engineering*. 84–93.

D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 48–61.

C.D. Manning, P. Raghavan, and H. Schütze. 2008. *Introduction to information retrieval*. Cambridge University Press. I–XXI, 1–482 pages.

C. McMillan, M. Grechanik, D. Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the International Conference on Software Engineering*. 111–120.

S. Meng, X. Wang, L. Zhang, and H. Mei. 2012. A history-based matching approach to identification of framework evolution. In *Proceedings of the International Conference on Software Engineering*. 353–363.

A. Mishne, S. Shoham, and E. Yahav. 2012. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. 997–1016.

H. A. Nguyen, T. T. Nguyen, G. Wilson Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. 2010. A Graph-based Approach to API Usage Adaptation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. 302–321.

M. Nita and D. Notkin. 2010. Using twinning to adapt programs to alternative APIs. In *Proceedings of the International Conference on Software Engineering*. 205–214.

D. Perelman, S. Gulwani, T. Ball, and D. Grossman. 2012. Type-directed completion of partial expressions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 275–286.

J. H. Perkins. 2005. Automatically generating refactorings to support API evolution. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Lisbon, Portugal, 111–114.

P. Persson. 2007. MIT 18.335: Introduction to Numerical Methods (Fall 2007). http://persson.berkeley.edu/18.335/. (2007). Accessed: 2012-07-30.

A. Podgurski and L. Pierce. 1992. Behavior sampling: a technique for automated retrieval of reusable components. In *Proceedings of the International Conference on Software Engineering*. 349–361.

S. P. Reiss. 2009. Semantics-based code search. In *Proceedings of the International Conference on Software Engineering*. 243–253.

M. Rittri. 1990. Retrieving library identifiers via equational matching of types. In *Proceedings of the International Conference on Automated Deduction*. 603–617.

M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. 2013. Automated API Property Inference Techniques. *IEEE Trans. Softw. Eng.* 39, 5 (May 2013), 613–637.

N. Sahavechaphan and K. Claypool. 2006. XSnippet: mining For sample code. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. 413–430.

A. Santhiar, O. Pandita, and A. Kanade. 2013. Discovering math APIs by mining unit tests. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*. 327–342.

T. Schäfer, J. Jonas, and M. Mezini. 2008. Mining framework usage changes from instantiation code. In *Proceedings of the International Conference on Software Engineering*. 471–480.

J. Stylos and B. A. Myers. 2008. The implications of method placement on API learnability. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 105–112.

C. Teyton, J. Falleri, and X. Blanc. 2012. Mining Library Migration Graphs. In *Proceedings of the Working Conference on Reverse Engineering*. 289–298.

S. Thummalapenta and T. Xie. 2007. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the International Conference on Automated Software Engineering*. 204–213.

J. Vesanto, J. Himberg, E. Alhoniemi, and J. Parhankangas. 2000. Self-Organizing Map in Matlab: the SOM Toolbox. In *Proceedings of the Matlab DSP Conference*. 35–40.

P. Weissgerber and S. Diehl. 2006. Identifying Refactorings from Source-Code Changes. In *Proceedings of the International Conference on Automated Software Engineering*. 231–240.

W. Wu, Y. Guéhéneuc, G. Antoniol, and M. Kim. 2010. AURA: a hybrid approach to identify framework evolution. In *Proceedings of the International Conference on Software Engineering*. 325–334.

Y. Ye and G. Fischer. 2002. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the International Conference on Software Engineering*. 513–523.

K. Yessenov, Z. Xu, and A. Solar-Lezama. 2011. Data-driven synthesis for object-oriented frameworks. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. 65–82.

K. Yuen. 2010. *Bayesian methods for structural dynamics and civil engineering*. John Wiley & Sons.

A. M. Zaremski and J. M. Wing. 1993. Signature matching: a key to reuse. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 182–190.

A. M. Zaremski and J. M. Wing. 1997. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.* 6, 4 (Oct. 1997), 333–369.

H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. 2010. Mining API mapping for language migration. In *Proceedings of the International Conference on Software Engineering*. 195–204.