Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 498618, 13 pages
http://dx.doi.org/10.1155/2015/498618

Research Article

ScalaLab and GroovyLab: Comparing Scala and Groovy for Scientific Computing

Stergios Papadimitriou,[1] Kirsten Schwark,[2] Seferina Mavroudi,[3,4] Kostas Theofilatos,[3] and Spiridon Likothanasis[3]

[1]*Department of Computer Engineering & Informatics, Technological Educational Institute of Kavala, 65404 Kavala, Greece*
[2]*iDashboards, 900 Tower Drive, Troy, MI 48098, USA*
[3]*Department of Computer Engineering and Informatics, University of Patras, Greece*
[4]*Technological Educational Institute of Patras, 26332 Patras, Greece*

Correspondence should be addressed to Stergios Papadimitriou; sterg@teikav.edu.gr

Received 26 January 2014; Revised 14 August 2014; Accepted 12 January 2015

Academic Editor: Damian Rouson

ScalaLab and GroovyLab are both MATLAB-like environments for the Java Virtual Machine. ScalaLab is based on the Scala programming language and GroovyLab is based on the Groovy programming language. They present similar user interfaces and functionality to the user. They also share the same set of Java scientific libraries and of native code libraries. From the programmer's point of view though, they have significant differences. This paper compares some aspects of the two environments and highlights some of the strengths and weaknesses of Scala versus Groovy for scientific computing. The discussion also examines some aspects of the dilemma of using dynamic typing versus static typing for scientific programming. The performance of the Java platform is continuously improved at a fast pace. Today Java can effectively support demanding high-performance computing and scales well on multicore platforms. Thus, both systems can challenge the performance of the traditional C/C++/Fortran scientific code with an easier to use and more productive programming environment.

1. Introduction

The recently introduced ScalaLab [1] scientific programming environment for the Java Virtual Machine (JVM) leverages the statically typed Scala object-oriented/functional language [2]. It provides a MATLAB-like syntax that is used to construct scripts that are then compiled by ScalaLab for execution on the JVM.

The GroovyLab environment is based on the Groovy dynamic language for the Java platform [3]. The underlying mechanisms in GroovyLab are very different from ScalaLab, primarily due to the dynamic character of Groovy.

The Scala language supports the implementation of simple, coherent, and efficient MATLAB-like interfaces for many Java scientific libraries. These interfaces are compiled within the core of ScalaLab. The Groovy language also provides mechanisms to access easily and elegantly many Java scientific libraries, but these mechanisms are quite different from those provided by ScalaLab.

ScalaLab and GroovyLab are open source projects and can be obtained from http://code.google.com/p/scalalab/ and http://code.google.com/p/jlabgroovy/, respectively. Both ScalaLab and GroovyLab can be installed easily. The only prerequisite is the installation of the Java 8 (or newer) runtime (which is free). We supply scripts for launching these systems for all the major platforms. Also, for Linux 64 bit, and Windows 64 bit platforms, native executables (e.g., for Windows the *WinScalaLab.exe*) provide an even easier startup. The general high-level architecture of ScalaLab is depicted in Figure 1 and is described in [1]. The architecture of GroovyLab is shown in Figure 2 and is very similar to that of ScalaLab, except GroovyLab uses the Groovy programming language rather than the Scala programming language. It is also a successor of jLab that is described in [4].

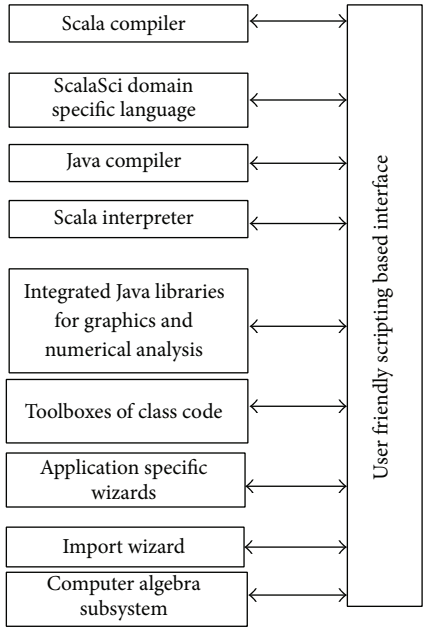

Figure 1: The architecture of the main software components of ScalaLab.

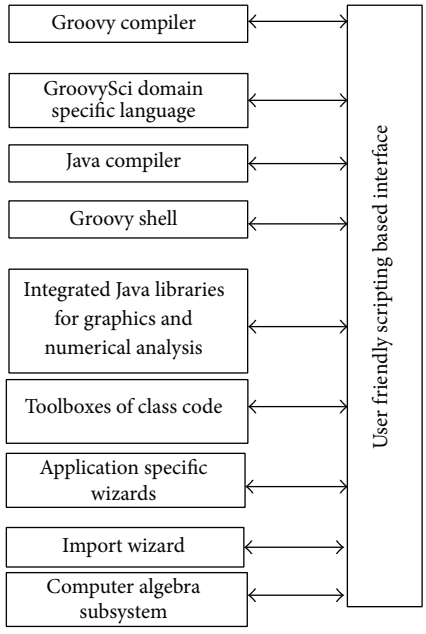

Figure 2: The architecture of GroovyLab.

GroovyLab is essentially a redesign of jLab that is based on the Groovy scripting language, which proves to be superior to the various scripting modes provided by jLab (j-script, JavaSci, GroovySci, compiled j-script mode). In a few words, the GroovySci scripting mode of jLab proved much superior and GroovyLab was developed by improving that mode and removing the others as somewhat redundant. Historically, ScalaLab was developed in parallel with GroovyLab as a similar environment using the powerful statically typed Scala language.

In this paper some important similarities and differences between ScalaLab and GroovyLab will be examined. The user interfaces of the two systems are similar. Also, the exploited Java scientific libraries and native code high performance libraries are the same. The major differences emerge when code is developed for these systems. They present different scripting languages for writing applications and are also very different when designing and implementing libraries.

The paper proceeds as follows: Initially the frameworks for developing matrix libraries in Scala and in Groovy will be examined (Section 2). Next the implementation of high-level mathematical operators in ScalaLab and GroovyLab is discussed (Sections 3 and 4). The functional programming abilities of the two environments are then briefly examined (Section 5). Both systems provide the user with flexible scripting environments; the main features of these environments are then compared (Section 6). Compile-time metaprogramming is a powerful feature offered both by Groovy and Scala; an example is then presented to demonstrate how compile-time metaprogramming can be used to expand the syntax of GroovyLab without any run-time performance penalties (Section 7). A few aspects of the ScalaLab and GroovyLab environments that are important for scientific computation are then presented and compared. Next performance related issues are discussed and benchmarking results are presented (Section 8). Finally, the paper concludes with remarks concerning the relative strengths and weakness of each system.

2. Matrix Design in ScalaLab and GroovyLab

In this section some main features of the Scala and Groovy languages that are used to facilitate the utilization of the Java scientific libraries are described and compared. These features are presented in the context of providing support for a MATLAB-like syntax for matrix manipulation and the utilization of underlying Java libraries to provide the implementation of the matrix functionality.

2.1. Matrices in ScalaLab. The general architecture for interfacing with Java libraries in ScalaLab is illustrated in Figure 3. Below we describe these components.

2.1.1. The Java Library. The *Java library* module in Figure 3 corresponds to the Java code of the library that performs the main numerical calculations. Some examples of the Java libraries are the EJML library (https://code.google.com/p/efficient-java-matrix-library/), the Apache Common Maths (http://commons.apache.org/proper/commons-math/), and the MTJ (Matrix toolkits for Java, https://github.com/fommil/matrix-toolkits-java). It should be noted that the Scala interpreter can also use the native Java interface of each library.

2.1.2. The Wrapper Scala Class (WSC). The *Wrapper Scala Class (WSC)* aims to provide a simpler interface to the more essential functionality of the Java library; for example, matrices A and B can be simply added as $A + B$, rather than invoking the cumbersome *A.plus(B)*. The wrapper Scala class

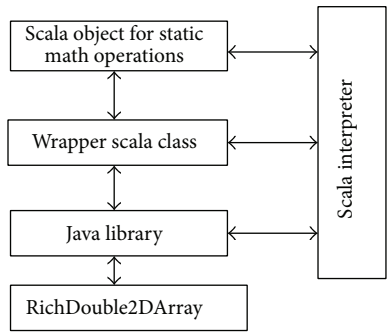

Figure 3: The general architecture of interfacing Java libraries in ScalaLab.

Matrix represents a one-indexed matrix and is based on the NUMAL library [5]. The wrapper class *Mat* is a zero-indexed matrix implemented in Scala. It borrows functionality from the JAMA Java package. Some other wrapper classes exist that interface the functionality of important Java libraries as, for example, the *EJML.Mat* class based on the EJML library and the Apache Common Maths library. We implement separate zero-based and one-based indexed *Matrix* classes for two reasons: (a) obtaining maximum implementation efficiency and (b) the one-indexed *Matrix* class is a wrapper for the NUMAL library routines. We feel that is rather inconvenient to mix zero and one indexing styles at the same *Matrix* class.

2.1.3. The Scala Object for Static Math Operations. The *Scala Object for Static Math Operations (SOSMOs)* provide overloaded versions of basic routines for our new Scala matrix types. For example, it provides an overloaded version of the *sin* method that accepts an instance of our *Mat* Scala class as an argument (i.e., *sin(B)* where *B* is a *Mat* instance).

Each SOSMO object implements a large set of mathematical operations. The rationale behind these objects is to facilitate the switching of the Scala interpreter to a different set of libraries. The interpreter simply needs to import the corresponding SOSMO objects in order to switch functionality.

The top-level mathematical functions for the zero-indexed matrices, for example, *rand0(int n, int m)*, *ones0(int n)*, and so forth, return the matrix representation associated with the currently utilized library. A matrix object can refer to different matrices depending on the library. The "switching" of libraries is performed by creating a different, new Scala interpreter that imports the corresponding libraries with the aid of the specially designed SOSMOs Scala objects. For example, the *StaticMathsJAMA* object performs important initializations for the JAMA library and the *StaticMathsEJML* utilizes the Efficient Java Matrix Library (EJML). The utilization of the JAMA library is accomplished by creating a Scala interpreter that imports the *StaticMathsJAMA* SOSMO object while for the EJML the *StaticMathsEJML* is imported. The ScalaLab user can easily switch different underlying Java libraries.

2.1.4. The RichDouble2DArray. The *RichDouble2DArray* is the "super" Matrix class of ScalaLab. It implements mathematical routines that expose the best aspects of various Java scientific computing libraries. It is however independent of any particular utilized library. By convention, utility routines that do not end in 0 or 1 return *RichDouble2DArray* objects. For example, *rand()*, *zeros()*, *ones()*, and so forth all construct *RichDouble2DArray* objects. Furthermore, the extensibility of *RichDouble2DArray* is leveraged with implicit conversions in order to provide its rich functionality to standard two-dimensional Java/Scala arrays.

2.2. Matrices in GroovyLab. The design of matrix support in GroovyLab is simpler than that in ScalaLab. Instead of providing switchable matrix interfaces to different libraries like ScalaLab, GroovyLab provides one powerful *Matrix* class that aims to combine effective numerical routines from multiple numeric libraries. In essence this *Matrix* class has many similarities in functionality to the *RichDouble2DArray* class of ScalaLab. As *RichDouble2DArray*, it provides a lot of operations, in pure Java for efficiency. Also, a set of efficient native code routines is interfaced with JNI (Java Native Interface) from the BLAS and LAPACK libraries.

The *Matrix* class of GroovyLab is a zero-indexed, two-dimensional dense matrix type that realizes much functionality of *GroovySci*. *GroovySci* is the scripting language of GroovyLab that is an extension of Groovy that provides MATLAB-like operators and syntax (corresponds to *ScalaSci* of ScalaLab). The Matrix class leverages functionality from multiple libraries such as JAMA, LAPACK, NUMAL, EJML, JBLAS, Apache Common Maths, and MTJ.

The *Matrix* class is fundamental in GroovySci because of the following.

(a) It provides a number of *mathematical operations* on the Matrix object that are implemented using a variety of Java libraries. For example, for linear system solvers, the solver from the JLAPACK library, the NUMAL library, or the JAMA library could be used.

Some libraries such as the *Apache Common Maths* library, the *JAMA* library, and the *NUMAL* library use a two-dimensional double array matrix representation. The GroovyLab *Matrix* class also uses the same underlying matrix representation; therefore their routines are readily accessible without any conversion. Some other libraries use different matrix representations. For example, JLAPACK uses a one-dimensional double array representation, in which the matrix storage layout is in column based order (i.e., Fortran like). In these cases, *O(N)* conversion routines (where *N* is the number of matrix elements) are required before using the methods of these libraries. However, mathematical routines with much higher complexity than simple linear complexity (e.g., matrix factorization, Singular Value Decomposition, eigenvalue computations) benefit from such libraries. The EJML library also uses a one-dimensional "flat" matrix representation, in either row or column based order. EJML is one of the more efficient pure Java numerical libraries, one reason being the proper setup for effective caching that the one-dimensional storage representation presents.

(b) It provides many useful *static methods* that are usually overloaded to handle many different types. For example, the *sin()* method is overloaded to handle input from a Matrix, a two-dimensional double array, and a one-dimensional double array:

Matrix sin(Matrix a)
double [] [] sin(double [] [] a)
double [] sin(double [] a)

Static importation of all the static methods of the Matrix class is performed by GroovyLab before any code is executed with the GroovyShell (the component that executes Groovy scripts); therefore we can write *sin(x),* where *x* can take many possible types, for example, *Matrix, double [][], double [],* and *double*. Therefore, the Matrix class provides much of the same functionality provided by the Scala Objects for Static Math Operations described in the previous section.

2.3. Sparse Matrices. ScalaLab and GroovyLab provide extensive support for sparse matrices using the Java implementation of the techniques presented in [6] (the source is supplied free from the authors). Both systems also implement classes that wrap the sparse routines with a higher level and elegant syntax. Clearly, GroovyLab exploits Groovy's facilities for building Domain Specific Languages (DSLs) and ScalaLab those of Scala. Also, the open source project *Matrix Toolkits for Java* (https://github.com/fommil/matrix-toolkits-java) offers an effective framework for handling sparse matrices with Java classes. These classes are integrated in the built-in libraries of both ScalaLab and GroovyLab and can be utilized with MATLAB-like convenience.

3. Designing High-Level Operators in ScalaLab and GroovyLab

Both Scala and Groovy provide the ability to define operators that function on operands of arbitrary types. Groovy's support for operator definition is more limited than Scala's and is restricted to a set of well-known symbolic operators.

3.1. Defining Operators in ScalaLab. ScalaLab's support for operator definition is best demonstrated with an example. The class used as an example for ScalaLab is the *EJML.Mat* class.

The *SimpleMatrix* class of the Efficient Java Matrix Library (EJML, http://code.google.com/p/efficient-java-matrix-library/) implements mathematical operations in an object-oriented way while maintaining the immutability of the operands. For example, to multiply matrix F and x the *mul* method on F can be invoked as in *F.mul(x)*. The Java-like method calls are not very elegant though. For example the matrix calculation $F\ P\ F' + Q$ is executed *as F.mult(P).mult(F.transpose()).plus(Q)* instead of the much clearer $F*P*F\sim + Q$ that is performed in ScalaLab.

The *scalaSci.EJML.Mat* (abbreviated *Mat*) class in ScalaLab wraps the EJML *SimpleMatrix* class and provides the Scala support for high-level MATLAB-like operations.

```
def apply(row: Int, col: Int) = {
     sm.get(row, col)
 }
```

Algorithm 1

In Scala, operators on objects are implemented as *method calls*, even for primitive objects like integers (i.e., *int* type). Although operators are treated syntactically and semantically as method calls, at the code generation phase the Scala compiler treats the usual arithmetic operators on primitive types by generating direct efficient bytecode. Thus, Scala mathematical expressions have speeds similar to those of Java. Operator characters are valid method names. A familiar operator symbol (e.g., "+") can be used to define a method that implements the operator. Infix operators are implemented as methods that have a single parameter that is of the type of the second operand. For example, $a*5$ corresponds to $a.*(5)$. *Prefix* operators such as +, –, !, ~ are implemented by prepending the word: *unary_*, to construct the corresponding method name. *Postfix* operators are implemented in Scala as methods that take no arguments. For example, implementing a method named ~ on the Matrix class could be used to indicate Matrix transposition. So, the transpose of A would be obtained using the ~ operator as $A\sim$.

Scala provides the *apply* method to provide support for a subscript-like operator for indexed object access that appears similar to both method invocation and array element access. For example for the *EJML.Mat* class, the apply method can be used to provide access to the *(row, col)* element of the matrix. The method is implemented, by calling the corresponding routine *get()* of the EJML library, as shown in Algorithm 1.

The method is then invoked as $M(i, j)$ to access the element of the matrix in the ith row and jth column of M. The *apply* method can also be overloaded in order to obtain a subrange of a matrix as well.

In a similar fashion, Scala provides the *update* method to support indexed object assignment. For the *EJML.Mat* class, the update method can be implemented to assign a value to a specific element of the matrix. The method is implemented as shown in Algorithm 2.

The *update* method is then invoked as $M(i, j) = 9.8$ to assign the number 9.8 to the i-jth element of M.

ScalaLab provides an implementation of the colon (:) operator, to support the MATLAB colon operator for creating vectors as the following example:

```
var t = 0::0.02::45.9
```

This expression returns a scalaSci.Vec type for t. To implement such syntax, *implicit conversions* are combined with the *token cells* approach [7, 8].

Methods with names ending with the ":" characters are invoked on their right operand, passing in the left operand. For example, the above expression is evaluated as (45.9.::(0.02)).::(0). Since the *Double* class does not have a :: method, it is implicitly converted to a *MATLABRangeStart* object. The *MATLABRangeStart* object retrieves the

```
def  update(row: Int, col: Int, value: Double): Unit = {
   sm.set(row, col, value)
 }
```

Algorithm 2

```
x = rand(50, 100)
row = 5; col = 2
xr = x[1..4+row, col..4*col]  // take a range
xrb = x[(1..20).by(2), (1..30).by(3)]  // like, x(1:2:20, 1:3:30)
x[(1..40).by(5), 1..2] = 44.
```

Algorithm 3

receiver's value (i.e., 45.9) with its constructor and stores it as the ending value of the range. The *MATLABRangeStart* class has a method named :: that processes the increment parameter (i.e., 0.02). Finally, the method :: creates a *MATLABRangeNext* object passing itself, that is, the *MATLABRangeStart* object, as the argument. The :: method of the *MATLABRangeNext* receives as a parameter the starting value of the range (i.e., 0). Therefore, it has all the information (i.e. start, increment, end) to construct and return the vector *t*.

MATLAB-like indexing/assignment is implemented easily by defining overloaded versions of *apply()* that operate on vectors. For example, to evaluate the expression *M(2::4::20, 3::2::100)*, the implicit conversions mechanism converts the arguments to vectors. The apply method is then invoked which will extract from the vector arguments the necessary start, step, and increment values.

3.2. Defining Operators in GroovyLab. In contrast to Scala, Groovy provides support for operator overloading only for a specific set of operator symbols. The positive side of this restriction is that we enforced to avoid unusual operator symbols, conforming to the familiar ones (e.g., "$*$" for multiplication, "$\ll$" for left shift, etc.). To support operator overloading, Groovy supplies a standard mapping of operator to implementing method. For example, the "+" operator corresponds to the *plus* method. From Java only these methods can be used, but from Groovy either the operators or their corresponding methods may be used.

We provide implementations of the Groovy operator methods in pure Java code for both efficiency's sake and to use them from Java as methods. Scala operators cannot be implemented directly in Java (at least without considering the internal details of the Scala compiler) because Java does not support operators as method names. It would also be difficult to call them. In order to call the methods corresponding to Scala operators from Java, the synthetic names that the Scala compiler creates for the operators must be used, which is very inconvenient.

Operator support in Groovy is supplied by implementing the appropriate corresponding method. For example, the *plus* method is implemented to provide the addition operator "+," the *minus method* is implemented to provide the subtraction operator "–," and the *multiply* method is implemented to provide the multiplication "$*$" operator. This operator method mapping approach is not as flexible as Scala's "methods as operators" approach that permits the user to define arbitrary symbols as operators. These predefined methods in Groovy can be implemented in Java and (as noted) are implemented in GroovyLab in Java for the sake of efficiency. The syntactic convenience of using the operator rather than the method is only applicable in Groovy though. For example, $x + 100$ is a valid expression in Groovy, where x is a Matrix object, but in Java matrix addition is performed using *x.plus*(100).

Another example of Groovy's syntactic elegance is demonstrated in the implementation of matrix indexing and assignment operators. The subscript indexing operator in Groovy is overloaded by implementing the *getAt()* method and the subscript assignment operator is overloaded with the *putAt()* method. Groovy has built-in support for integer ranges via the .. operator, so the syntax 2..5 can be used to create an *IntRange* instance. The concept of a step can be implemented by defining a class named *IntRangeWithStep* that inherits from the Groovy range class *IntRange*. This class is used in GroovyLab to write elegant MATLAB-like constructs, as shown in Algorithm 3.

Since the predefined *IntRange* type does not have a *by()* method, the Groovy compiler handles 1..20 as *IntRangeWithStep*. The *by()* method stores the step argument. Finally, elements are accessed using the subscript access operator implemented with the *getAt()* method and the subscript assignment operator that is implemented by the *putAt()* method.

Similarly, the *DoubleRangeWithStep* class that extends *ObjectRange* is provided by *GroovyLab and* implements the *step()* method in order to return a vector. Thus we can write

x = (0.5..50).step(0.01) as equivalent to MATLAB's *0.5:0.01:50*.

After examining both Scala's and Groovy's support for operator overloading, it can be concluded that while GroovyLab's

syntax is convenient it cannot provide the MATLAB-like syntax that can be implemented in ScalaLab.

3.3. Matrix Operations Performance. The performance of the basic indexing and assignment operations in GroovyLab's Matrix class when Groovy is statically compiled is similar to that of Scala's two-dimensional matrices. Other Matrix range operations are slower in GroovyLab, even though many of the submatrix operations are performed in Java. Specifically, for operations involving large submatrices the speed is about the same, but performance is about three times better in ScalaLab than in GroovyLab when many small submatrices are processed. In the latter case, the GroovyLab implementation involves much more dynamic Groovy code, rather than the faster Java code, hence the performance penalty.

4. Defining Operators for User Types: Implicit Conversions versus the Metaobject Protocol

Both Scala and Groovy provide run-time mechanisms to handle the situation where an operator is applied to a type in which there is not a specific operator that matches the argument types. Implicit conversion of argument types is supplied in Scala to address these potential type mismatches. Groovy provides metaprogramming to address these sorts of type violations.

4.1. Implicit Conversions in Scala. Returning to the matrix *Mat* class example in Scala, when the compiler detects the addition operator "+" on a *Double* object *d* that adds a *Mat* object *M*, that is, $d + M$, it encounters a type error because there is no method defined on the predefined *Double* type that adds a *Mat* instance to a *Double* (and there cannot be one since *Mat* is a user defined type). A similar error occurs when a Mat instance is added to a double array.

Scala provides implicit conversions [2, 9, 10] to address these sorts of type issues. When an operation is not defined for a type, the compiler will try to apply available implicit conversions in order to transform the type into a type for which the operation is valid.

The concept of implicit conversion is of fundamental importance in the construction of high-level mathematical operators in ScalaLab. Implicit conversion in ScalaLab is used with many classes, for example, *RichNumber, RichDouble1DArray*, and *RichDouble2DArray* classes.

For example, the *RichNumber* class is implemented to support implicit conversions in Scala related to the Double class. The RichNumber class models extended Number capabilities of accepting operations with all the relevant classes of ScalaLab, for example, with *Mat, Matrix, EJML.Mat, MTJ.Mat,* and generally whatever class we need to process.

Suppose that we have

var a = 2.0 + rand(2,2)

The 2 is transformed by the Scala compiler to a *RichNumber* object that defines an operation to add a Matrix and the addition can be performed by the addition operator implementation.

Similarly, the classes *RichDouble1DArray* and *RichDouble2DArray* wrap the *Array[Double]* and *Array[Array*[*Double]]* Scala classes in order to support implicit type conversion for the addition and multiplication of *Array[Array[Double]]* types.

As *RichNumber* enriches simple numeric types, *RichDouble1DArray* enhances the *Array[Double]* type and *RichDouble2DArray* the *Array[Array[Double]]* type. For example, the following code is valid in Scala:

```
var a = Ones(9, 10) // an Array[Array
[Double]] filled with 1s
var b = a+10 // add the value 10 to
all the elements returning b as
RichDouble2DArray
var c = b + a⋆89.7 // similarly using
implicit conversions this computation
proceeds normally
```

In the next section we continue by describing the corresponding implementations of high-level mathematical operators in the context of GroovyLab. Although similar functionality as in ScalaLab can be achieved in GroovyLab, the underlying approaches are very different.

4.2. The Metaobject Protocol in Groovy. In Groovy as in many other dynamic languages, the implementation of high-level mathematical operators for the standard language types is based on the *Metaobject protocol. The Metaobject protocol* forms the basis of metaprogramming in Groovy that is used to implement dynamic method invocation [3]. This protocol is the means by which dynamic functionality can be added to classes at runtime. Dynamic behavior is added to classes and objects in Groovy using the *MetaClass* machinery.

In dynamic languages, methods can be injected into a class by adding methods to its *MetaClass*. These added methods are then available globally on instances of the class. In the case of Groovy, metaprogramming can be used to add methods, properties, constructors, and static methods to classes at runtime. New methods in Groovy can be added to both Groovy classes and Java classes. Groovy supports metaprogramming at both the class and object level.

For example, the *Number* class of the standard Groovy's library does not implement the addition operator to support adding an array to a Number instance. Metaprogramming can easily be used to define a method on the number class to support this addition operation and adding it to the MetaClass of the Number class, the Groovy Code is illustrated in Algorithm 4.

The keyword *delegate* refers to the current object, that is, the *Number* object. Also, since Groovy's bytecode is somewhat slow for numeric calculations, GroovyLab intermixes Java code to attain improved performance.

Although this mechanism is seemingly easier to implement than Scala's implicit conversions, it requires indirect

```
//  define the operation: Number + double [],
// at the MetaClass of the Number's class
Number.metaClass.plus =
    {
//  the double []  m array denotes the input parameter
  double []  m ->
// call a Java routine for the operation
     res =
groovySci.math.LinearAlgebra.LinearAlgebra.plus(delegate, m)  // calls Java code
       res
}
```

Algorithm 4

method calls through the MetaClass machinery and imposes additional method invocation overhead. However, Groovy's approach is more flexible in that it supports changing which method will be executed at runtime. This flexibility is useful if implementations from different libraries are used interchangeably, for example, when it is desired to use an eigenvalue decomposition from a different library than the one supplied by default.

5. Functional Programming: Functions versus Closures

A core concept in any language that supports functional programming is the provision of functions as "first-class citizens." In object-oriented languages like Scala and Groovy functions are objects. They can be passed as arguments to other functions, be a return value from a method, and have a concise literal definition syntax. Most programming languages now support first-class functions because they vastly improve the readability and understandability of the source code by allowing behavior to be captured, assigned to variables, and passed to functions. For instance, the major new feature of Java 8 is the support of functional programming with *lambda expressions*.

5.1. Functions in Scala. Scala has the concept of both *functions* and *closures*. Functions are like static methods that do not belong to any class/object.

An example of a function definition in Scala is

$$def\ cube(x: Double) = x*x*x$$

Scala also has a literal function syntax that can be used to assign a function to a variable. For example, the function above can be declared as follows:

$$val\ cube = (x: Double) => x*x*x$$

The Function variable "cube" can be invoked as if it were the function definition above that is as in the invocation *cube(3)*.

A function in Scala that refers to a variable outside of its scope is called a *closure*. These variables from outside the scope of function are referred to as *free* variables. An example of the definition of a Scala closure is

$$val\ raiseIt = (x: Int) => Math.pow(x, power)$$

Here "power" is a free variable because it is defined outside of the scope of the function. At compile time a variable named "power" must be in the scope of the literal definition of the *raiseIt* function; otherwise a compiler error will result.

Therefore, Scala differentiates between a function and a closure based on whether or not the definition contains free variables.

5.2. Closures in Groovy. Groovy also supports "global" function definitions (i.e., functions that are not defined within a class) but these functions are simply static methods of classes imported automatically. Therefore, they do not support the functional programming style.

Groovy however provides strong support for closures that are *first-class* objects that can be passed as parameters and assigned to variables. The syntax of closure definition is different from the definition of methods; for example, a simple Groovy closure that implements the *cube* function is

$$def\ cube = \{x -> x*x*x\}$$

The closure is then invoked as expected as *cube(3)*.

5.3. Global Function Workspace. Scientific programming environments demand a *global namespace of functions*. Scala and Groovy do not have the concept of globally visible methods; every method must be contained in an object or a class. In both environments though, a global function namespace can be implemented easily with *static imports*. In Scala objects are imported since these objects encapsulate the static imports. In Groovy static imports are performed as they are in Java. Therefore, the automatic import of static methods provides the appearance of the existence of global methods. For example, the *plot* method appears to be available globally since we import it from the object *scalaSci.plot.plot*. Scala also offers the ability to define *apply* methods for the companion objects of classes. If a class implements the apply method, an instance of the class can essentially be "executed" like a function as the instance name followed by a list of arguments in parentheses. When the apply method is implemented by a class, a method does not need to be imported into the "global" namespace, it is only necessary to import the class itself. Groovy does not offer a similar ability to essentially execute

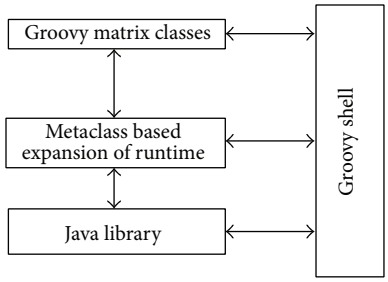

Figure 4: The general architecture of interfacing Java libraries in GroovyLab.

object instances, but the classic *import static* Java facility can be used to import the appropriate static methods.

6. The Scala Interpreter and the Groovy Shell

An essential component of Scala's scripting framework is the Scala interpreter. The corresponding component in the Groovy scripting framework is the GroovyShell. This section discusses and compares some aspects of these components (Figure 4).

The overall approach of the Scala interpreter is based on initially compiling the requested code. A Java classloader and Java reflection are then used to run the compiled bytecode and access its results.

In general, Scala's approach to scripting and Groovy's approach to scripting are similar. Scala's approach is more complicated than Groovy's approach though. Groovy's scripting approach is based on detecting the undeclared variables of a script. Groovy then declares them implicitly as an Object and maintains their state in a *binding structure* implemented by means of a Java hashtable. The binding scheme of Groovy is restricted to data variables and closures. It is important that although GroovyShell does not keep objects and classes, it keeps *closure* objects. Therefore, the computation workspace consists not only of data variables but also of defined code. This permits functional programming within GroovyLab.

In contrast, the scheme implemented in the Scala interpreter extracts the whole visible computation state as the interpreter's context. ScalaLab binds both data and code objects automatically to this context. Although this is more powerful it imposes difficulties in retrieving the context when we create a new Scala interpreter. In that case it is necessary to replay the commands in order to restore the environment. However, the restoration of the user environment in ScalaLab is performed fast; the user does not notice the delay of the few Scala compiler runs. Restoration of the computation context also is a somewhat rare operation.

A single compiler instance is used in the Scala's interpreter to accumulate all successfully compiled or interpreted Scala code. The interpretation of a line of code is performed by generating a new object that includes the line of code and has public members to export all variables defined by that code.

The results of the interpreted lines are extracted by using a second "result object" which imports the variables exported by the above object and then exports a single member named "result." To accommodate user expressions that are read from variables or methods defined in previous statements, "import" statements are used.

It becomes evident that an effective approach for detecting the variables that a piece of code defines is required. The Scala interpreter utilizes the Scala parser to accomplish the nontrivial task of analyzing variable visibility. The Scala parser is also utilized, in order to detect which variables are used by a code snippet. The values of these variables are then requested from previously executed code. It is unnecessary to request variables from the interpreter that do not appear in a new code snippet, since these values will not change. For example, if *prevVar* = 5.5 is not used at our new script, the interpreter does not request the value of prevVar.

At this point we should contrast the state management of the Scala interpreter with the corresponding one of the GroovyShell. GroovyShell automatically handles variable binding and therefore it is much easier to handle the state of variables in GroovyLab.

The Scala interpreter keeps a list of names of identifiers bound with a var or val declaration (*boundVarNames*) and also a list of function identifiers defined with a def declaration (*boundDefNames*). In addition we can retrieve the last text source code of the program that has been interpreted (*lastLineInterpreted*).

We keep a symbol table of the current *ScalaSci* bound variables. This task is used to graphically display the current work context to the user. We have to keep this external table synchronized with the internal variables binding that the interpreter keeps (i.e., *boundVarValNames*). The current value of each variable is retrieved from the interpreter by issuing a simple command with the name of the variable.

If an identifier is being redefined as a function then it is removed from the variable binding. This is necessary since the namespace of variables and functions in Scala is common [2] and thus the identifier is hidden by the function definition.

The corresponding task in the GroovyShell of displaying to the user the variable's workspace is much simpler; in fact it only requires a simple lookup into the hashtable binding.

The GroovyShell lacks the ability to retain imports from a previously executed script. This can be rather inconvenient in many cases, since these same import statements must be manually incorporated before executing code that depend on them. For that reason, GroovyLab implements simple import buffering that removes the tediousness of incorporating the import statements explicitly each time a script is executed in GroovyLab.

GroovyLab automatically incorporates some basic import statements into the user's scripts that extend the functionality. These imports allow the user to utilize many GroovyLab functions such as figure() and plot(). The user can also use a code buffer to retain code (both classes and script code) for the current working session.

7. Compile-Time Metaprogramming

Compile-time metaprogramming allows intervening in the compilation process and the implementation of customized

structures that do not impose any run-time overhead. This is important in scientific scripting, since it opens up possibilities for implementing efficient convenient high-level structures.

Both languages support compile-time metaprogramming, Groovy with Abstract Syntax Tree (*AST*)—*Transformations* and Scala with *Scala Macros. Abstract Syntax Trees (AST)* keep a tree like representation of a program in which interior nodes represent programming constructs [11]. Scala macros have not been utilized in ScalaLab yet (actually they are used only to provide a C style *for* loop), so the discussion of compile-time metaprogramming will be limited to an exploration of Groovy's compile-time metaprogramming support.

Compile-Time Metaprogramming in Groovy. Run-time metaprogramming provides the ability to modify class and instance behavior at runtime. Groovy provides compile-time metaprogramming via *AST transformations* and *compilation customizers* to support altering the behavior of classes and objects at compile time.

AST transformations in Groovy come in two flavors: *local AST transformations* and *global AST transformations*. Local AST transformations are defined using annotations and are applied locally to a program element by annotating the element whose AST is to be altered by the AST transformation. Global AST transformations are essentially provided as services that are applied by the compiler to every class that is being compiled. Local AST annotations are more widely used than global AST transformations.

Groovy implements some predefined local AST transformations. Examples include the @Immutable AST transformation that generates code to make a class immutable and the @Log AST transformation which injects a logger into a class. Of specific interest to GroovyLab is the *@CompileStatic AST transformation*. This AST transformation is applied to Groovy code that is known not to utilize Groovy's dynamic programming capabilities. When the @CompileStatic AST transformation is applied to a method the compiler checks to see that the source code can indeed be statically compiled and bypasses Groovy's dynamic dispatch for methods that are invoked within the method implementation. This ability to bypass dynamic dispatch can have a marked impact on performance, which is of the utmost importance in scientific computing. We should note that the recent versions of the Groovy compiler automatically avoid dynamic dispatch in simple numerical loops; thus in such cases Groovy's performance is similar to Java/Scala, even without the static compilation annotation.

GroovyLab also provides the *@CompileJava AST transformation*. This transformation is applied only to methods and essentially compiles the method directly into Java bytecode. The code is compiled as it appears in the Groovy source code, so the source code of the annotated method must be valid Java and must be completely self-contained (i.e., it must essentially be a static method that contains no references to class attributes). The @CompileJava AST transformation provides the ability to execute Java code from within a GroovyLab script, both for efficiency and as a preferred choice.

Groovy also provides the ability to modify the behavior of the compiler using compilation customizers. Compilation customizers are applied by the compiler to the AST corresponding to every class that is being compiled. They differ from global AST transformations in that customizers are easier to define than global AST transformations. GroovyLab uses a compilation customizer to convert BigDecimal literals to primitive double to improve performance.

8. Performance

Groovy was initially considered a slow language; Groovy 1.5 was about 200 to 1000 times slower than Java in performing number calculations. The implementers of the language have performed impressive performance improvements in recent versions of Groovy though. A clever implementation of call-site caching in Groovy 1.6 has reduced the performance gap between Groovy and Java to Groovy being about 20–50 times slower than Java. Special handling of primitive operations as direct bytecode rather than via dynamic dispatch using the MetaObject protocol has provided significant performance gains, primarily because these optimizations can be applied at compile time. In many cases, Groovy with compiled computational loops and primitive operations runs as fast as Java code (since both emit direct bytecodes). Statically compiled Groovy, a mode of the Groovy compiler introduced with Groovy 2.0, produces statically compiled code that is as fast as Java but requires sacrificing the dynamic features of the Groovy language in the source code that is to be statically compiled. Therefore, this mode is most suitable for computationally intensive methods. Groovy 2.0 also exploits the JDK7 *invoke-dynamic* bytecode and the related framework that supports the compilation of dynamic code. However in the current release of Groovy 2.4 the invoke dynamic implementation improved a lot related to Groovy 2.0. Generally code that is compiled with Groovy's invoke dynamic support currently runs somewhat slower than the code that optimizes primitive operations. The speed difference is constantly decreased on each new Groovy release. As both the invoke dynamic support in Groovy improves and Java Virtual Machine implementations better support the invoke dynamic instruction, the static compilation and primitive optimizations features of Groovy become of less importance.

GroovyLab is designed with the goal in mind of performing fast numeric calculations. For example, all of the main mathematical operations in GroovyLab are implemented in pure Java for the sake of efficiency. The important *Matrix* class is also implemented in Java, but it implements the *GroovyObjectSupport* interface, in order to allow flexible overloaded operator syntax, for example, to use $A + B$ instead of something like the cumbersome method *A.plus(B)* to add matrices. The Matrix class also supports many of operations that make use of the native BLAS. This is accomplished with the JBLAS library (http://mikiobraun.github.io/jblas/).

Table 1

	ScalaLab	MATLAB	SciLab	GroovyLab
Speed	Very fast, execution speed depends on the Java runtime, generally faster than MATLAB at script code, but slower for routines implemented as built-in with MATLAB	Very fast, especially the built-in routines which are highly optimized; overall ScalaLab and MATLAB run at comparable speeds and which one outperforms depends on the case	Much slower than ScalaLab (or MATLAB), about 20 to 100 times slower. Newer versions of SciLab, however, improved a lot; speed differences are now about 3 to 10 times	Slower than ScalaLab, about 2 to 5 times slower. However, with statically typed blocks of code, performance is at about the same level as Java/Scala
Portability	Very portable, anywhere exists installed Java 8 JRE	There exist versions for each main platform, for example, Windows, Linux, MacOS	There exist versions for each main platform, for example, Windows, Linux, MacOS	Very portable, anywhere exists installed Java 8 JRE
Open source	Yes	No	Yes	Yes
User-friendliness	Very user friendly	Very user friendly	Very user friendly	Very user friendly
Libraries/toolbox availability	All the JVM libraries	A lot of toolboxes are available, but generally not free	There exist toolboxes for basic applications but for specialized ones it is difficult to find	All the JVM libraries
Documentation	Little yet, and limited to on-line help, since even main code components are in the development process	Extensive documentation	Sufficient documentation	On-line documentation only
Flexibility of the language (i.e, syntax malleability)	The Scala language is designed to be flexible and with very malleable syntax	The syntax of MATLAB is not designed to be extensible	SciLab is not designed to be extensible	The Groovy language is dynamic and different tricks from the Scala's case can form customizable syntax
Development of large applications	Scala has a lot of novel features that can facilitate the development of large applications. ScalaLab applications can run standalone, as any Java code	The notion of MATLABPATH integrates many MATLAB scripts, something not very scalable	Similar to MATLAB, the SciLab scripts are not well suited for complex applications, but rather they fit well for rapid testing of scientific algorithms	Groovy has a full compiler that can be used to produce standalone code of a large application project
Active user development community	ScalaLab is a new project, and thus up-to-now lacks a large user base	MATLAB has a huge user base	SciLab has a large user base, however, much smaller than MATLAB's	GroovyLab is a new project, and thus up-to-now lacks a large user base

Another performance pitfall with Groovy is the preference of the Groovy compiler to perform default mathematical calculations with *BigDecimal*s objects. Groovy is a flexible, extensible language that allows GroovyLab to bypass that constraint by performing a *compile-time metaprogramming* transformation. These transformations are a powerful tool to modify the AST representation of the source code that is generated during compilation, before bytecode is generated from it for execution. Since the AST modifications are performed by the compiler during compilation, there is no run-time performance penalty that results from applying the AST modifications.

In case of ScalaLab, the Scala language is statically typed and therefore Scala code can theoretically be compiled to bytecode that runs as fast as Java, sometimes a bit faster sometimes a bit slower, depending on the situation. However, for the advanced features of Scala such as pattern matching, trait inheritance, and type parameters, it is difficult to optimize their compilation. The Scala language developers concentrate on these issues and improve the performance of the Scala compiler with each new version of the language.

Table 1 compares characteristics of *ScalaLab, MATLAB, SciLab*, and *GroovyLab*. SciLab (http://www.scilab.org/) is an open source system, similar to MATLAB.

We would note that the performance of the recent version of MATLAB (2012b) has been impressively improved, while SciLab has been improved also but not so much.

9. Benchmarking

In order to access the performance of the GroovyLab and ScalaLab platforms, a variety of mathematical computation algorithms will be examined. These will include matrix computations, Fast Fourier Transforms (FFT), eigen decomposition of a matrix, and singular value decomposition of a matrix.

```
N = 2000; M = 2000
tic
a = rand(N, M);
sm = 0.0;
for r = 1 : N,
  sm = 0.0;
  for c = 1 : M,
    a(r, c) = 1.0/(r + c + 1);
    sm = sm + a(r, c) − 7.8 ∗ a(r, c);
  end
end
tm = toc
```

Algorithm 5: Array access benchmark in MATLAB.

9.1. Matrix Computation Benchmarking. In order to access the efficiency of matrix processing in GroovyLab and ScalaLab, implementations of the MATLAB script of Algorithm 5 are used.

For the ScalaLab version of this script, ScalaLab clearly outperforms both MATLAB and SciLab. GroovyLab has similar speed when the static compilation is used. With the implementation of *optimized primitive operations* (i.e., later versions of Groovy produce fast code for arithmetic operations since they avoid the overhead of the metaobject protocol) and with the later *invoke dynamic* implementation, Groovy generally is slightly slower than Scala. The reason for the superiority of ScalaLab in terms of scripting speed is clearly the statically typed design of the Scala language that permits the emission of efficient Java bytecode.

9.2. Fast Fourier Transform Benchmark. The Fast Fourier Transform (FFT) benchmark is performed in ScalaLab using implementations of the FFT from various libraries.

Of these libraries, the Oregon DSP library provides the best performance. Close in performance to this library is the JTransforms (https://sites.google.com/site/piotrwendykier/software/jtransforms) library. Since JTransforms is multithreaded and it will accordingly perform more efficiently with more robust machines (e.g., having 8 or 32 cores, instead of only 4). The tutorial FFT implementation of the classic Numerical Recipes book [12] (with the C/C++ code translated to Java) was also observed to achieve reasonable performance in ScalaLab. Interestingly, it was observed that the Oregon DSP and JTransforms FFT routines are nearly as fast as the optimized built-in FFT of MATLAB. We should note that the reported differences in benchmarks are stable; for example, the relative differences are about the same on different computers, and individual runs show small variations at the results. Contributing to the small variability is that we perform explicitly garbage collection before any benchmark run.

9.3. Other Benchmarks. Other types of problems such as the eigen decomposition, singular value decomposition, and solution of overdetermined systems were examined for the purposes of obtaining GroovyLab and ScalaLab benchmarks. The general conclusion is that ScalaLab is faster than SciLab 5.21 by about 3 to 5 times but is slower than MATLAB 7.1 by about 2 to 3 times. It is also evident that the routines of JLAPACK for special matrix categories run orders of magnitude faster than routines for general matrices; for example, for a 1500 by 1500 band matrix with 2 bands above and 3 bands below the main diagonal, the JLAPACK's SVD routines run about 250 times faster than for a general 1500 by 1500 matrix.

Table 2 summarizes some basic performance results. We should note that often ScalaLab and GroovyLab perform equally well since they call the same Java library routines.

Recent MATLAB versions have improved impressively the performance of Matrix multiplication and of many important routines, as, for example, the SVD computation. However, both ScalaLab and GroovyLab offer the potential to issue commands to the MATLAB engine using the Java/MATLAB interface. Similarly, SciLab scripts can be executed using the Java/SciLab interface. The wiki pages of the ScalaLab and GroovyLab projects describe details and provide examples of these interfacings.

9.4. Native Code Optimizations. In order to test the JVM performance versus native code performance, an implementation of SVD is used [see http://code.google.com/p/scalalab/wiki/ScalaLabVsNativeC]. Both the Microsoft's *cl* compiler of Visual Studio on Windows 8 64-bit and the *gcc* compiler running on Linux 64-bit were used. ScalaLab is based on the Java runtime version: 1.7.0_25 and Scala 2.11 M7, and GroovyLab on Groovy 2.2.1. Again both ScalaLab and GroovyLab perform similarly, since they are based on the same Java code. It has been observed that ScalaLab and GroovyLab perform better than unoptimized C and are even close to optimized C code when performing matrix calculations. Table 3 shows some results.

10. Conclusions and Future Work

This paper compares some aspects of ScalaLab and GroovyLab, which are both environments for scientific computing that run within the Java Virtual Machine framework. It was demonstrated that both environments can effectively utilize existing Java scientific software. Both can elegantly integrate well-known Java numerical analysis libraries for basic tasks. These libraries are wrapped by either Scala objects in ScalaLab or Groovy objects in GroovyLab and their basic operations are provided to the user with a uniform MATLAB-like interface.

An extension of Scala with MATLAB-like constructs called ScalaSci is the language of ScalaLab and the corresponding language of GroovyLab is GroovySci. Both languages are effective and convenient for both writing small scripts and for developing large production-level applications.

The design of the user interface of ScalaLab and GroovyLab is similar. Both emphasize user friendliness and provide integrated development environment- (IDE-) like features

Table 2: Results of some basic benchmarks.

	ScalaLab (secs)	SciLab 5.21 (secs), SciLab 5.5	MATLAB 7.1 (secs) MATLAB 2012b	GroovyLab (secs)
Matrix multiplication with matrix sizes: (2000, 2500) × (2500, 3000)	0.9 secs using Native BLAS combined with Java multithreading	61.8, 5.05	13.05, 0.6	The same with ScalaLab
LU				
1000	0.3	3.13, 2.42	0.36, 0.03	The same as ScalaLab
1500	1.2	3.82, 2.1	1.18, 0.04	As ScalaLab
2000	2.9	6.42, 1.6	2.72, 0.09	As ScalaLab
inv				
1000	2.7	12.97, 1.6	1.3, 0.05	As ScalaLab
1500	7.8	13.14, 2.5	4.5, 0.15	As ScalaLab
2000	9.31	19.07, 3.2	5.9, 0.3	As ScalaLab
QR				
1000	1.03	4.3, 4.2	1.2, 0.04	As ScalaLab
1500	3.7	9.96, 9.9	4.26, 0.2	As ScalaLab
2000	9.25	19.69, 19.3	9.89, 0.3	As ScalaLab
Matrix access scripting benchmark	0.03	32.16, 32.67	10.58, 0.32	0.031 static compilation, 0.156 with primitive ops, 0.211 with invoke dynamic
FFT 100 ffts of 16384 sized signal	**Oregon DSP:** real case: 0.05, complex case: 0.095 **JTransforms:** real case: 0.07 complex case: 0.11, **Apache Common Maths:** complex case: 0.5 **Numerical Recipes (Java Translation):** real case: 0.09 complex case: 0.12	Real case: 2.32 Complex case: 4.2	Real case: 0.05 Complex case: 0.08	The Java libraries for FFT are the same as ScalaLab's

Table 3: SVD performance: Java versus native C code.

Matrix size	Optimized C (gcc, similar is for cl)	ScalaLab/GroovyLab	Unoptimized C (gcc, similar is for cl)
200 × 200	0.08	0.15	0.34
200 × 300	0.17	0.2	0.61
300 × 300	0.34	0.58	1.23
500 × 600	3.75	5.06	8.13
900 × 1000	35.4	51.3	53.3

such as on-line help, code completion, graphical control of the class-path, and a specialized text editor with code coloring facilities that greatly facilitate the development of scientific software.

Future work will concentrate on improving the interfaces to Java basic libraries and on incorporating smoothly other interested libraries (e.g., the parallel COLT library for basic linear algebra, the JCUDA library for supporting the CUDA massively parallel computing framework on NVIDIA graphics cards). Both ScalaLab and GroovyLab explore the *symja* Java Computer Algebra system (https://code.google.com/p/symja/). This system implements a wide range of Computer Algebra facilities. Further work is in progress for making work with Computer Algebra easier. This work will include providing better on-line help and code completion for these routines. These components are of the utmost importance in incorporating this rather complicated libraries into ScalaLab and GroovyLab.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] S. Papadimitriou, K. Terzidis, S. Mavroudi, and S. Likothanassis, "ScalaLab: an effective scientific programming environment for the Java Platform based on the Scala object-functional language," *IEEE Computing in Science and Engineering*, vol. 13, no. 5, Article ID 5487486, pp. 43–55, 2011.

[2] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*, Artima, 2008.

[3] D. König, A. Glover, P. King, G. Laforge, and J. Skeet, *Groovy im Einsatz*, Manning Publications, 2007.

[4] S. Papadimitriou, K. Terzidis, S. Mavroudi, and S. Likothanassis, "Scientific scripting for the java platform with jlab," *Computing in Science and Engineering*, vol. 11, no. 4, pp. 50–60, 2009.

[5] H. T. Lau, *A Numerical Library in Java for Scientists and Engineers*, Chapman & Hall/CRC, Boca Raton, Fla, USA, 2003.

[6] T. A. Davis, *Direct Methods for Sparse Linear Systems*, SIAM Publishing, Philadelphia, Pa, USA, 2006.

[7] G. Dubochet, "On Embedding domain-specific languages with user-friendly syntax," in *Proceedings of the 1st Workshop on Domain Specific Program Development*, pp. 19–22, Nantes, France, July 2006.

[8] G. Dubochet, *Embedded domain-specific languages using libraries and dynamic metaprogramming [Ph.D. thesis]*, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2011.

[9] D. Wampler and A. Payne, *Programming Scala*, O'Reilly, 2009.

[10] V. Subramaniam, *Programming Scala: Tackle Multi-Core Complexity on the Java Virtual Machine*, Pragmatic Bookself, 2009.

[11] A. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques & Tools*, Addison-Wesley, Boston, Mass, USA, 2nd edition, 2007.

[12] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing.*, Cambridge University Press, Cambridge, UK, 2002.