Scientific Programming 22 (2014) 187–199
DOI 10.3233/SPR-140378
IOS Press

MATLAB-like scripting of Java scientific libraries in ScalaLab

Stergios Papadimitriou [a,*], Seferina Mavroudi [b,c], Kostas Theofilatos [b] and Spiridon Likothanasis [b]

[a] *Department of Computer and Informatics Engineering, Technological Educational Institute of Eastern Macedonia & Thrace, Kavala, Greece*
E-mail: sterg@teikav.edu.gr

[b] *Department of Computer Engineering and Informatics, University of Patras, Greece*
E-mails: {theofilk, likothan, mavroudi}@ceid.upatras.gr

[c] *Department of Social Work, School of Sciences of Health and Care, Technological Educational Institute of Patras, Greece Technological Educational Institute of Patras, Patras, Greece*

Abstract. Although there are a lot of robust and effective scientific libraries in Java, the utilization of these libraries in pure Java is difficult and cumbersome, especially for the average scientist that does not expertise in software development. We illustrate that ScalaLab presents an easier and productive MATLAB like front end. Also, the main strengths and weaknesses of the core Java libraries of ScalaLab are elaborated. Since performance is of paramount importance for scientific computation, the article discusses extensively performance aspects of the ScalaLab environment. Also, Java bytecode performance is compared to native code.

Keywords: Java, Scala, functional programming, scripting, interpreters, MATLAB, scientific programming

1. Introduction

Scripting environments are very popular in scientific computing especially for educational and prototyping applications. MATLAB dominates as a commercial scientific scripting package. Recently, we introduced the Scala based ScalaLab [16] environment for the Java Virtual Machine. ScalaLab exploits the powerful Scala object-functional language [12]. It presents a MATLAB-like style of working, and compiles the scripts for the JVM. ScalaLab runs the scripts at the full Java's speed which as we illustrate can be significantly better than unoptimized C code.

Three styles of programming coexist and can be combined in ScalaLab:

(a) *The MATLAB-like scripting*. It is the easiest one and the more convenient for small scientific programs.
(b) *The Java-like object oriented style*. It is rather inconvenient and verbose but fits well with application domains where inheritance can be effective in factoring common code.
(c) *The functional programming style*. It is very expressive but rather delicate. The articles [7,11] highlight many aspects of the functional programming style in scientific computing.

ScalaLab is an open-source project and can be obtained from http://code.google.com/p/scalalab/. It's top level architecture is depicted with Fig. 1. The language of ScalaLab is the *scalaSci*, an extension of Scala with MATLAB like syntactic constructs that implements the core scientific classes. These classes can be used also in standalone applications for the Java Virtual Machine. The present article focuses on the *scalaSci* domain specific language component. The *Scala Compiler* cooperates with the *Scala Interpreter* for the implementation of a flexible and fast compiled scripting framework. The *Java Compiler* can compile and execute pure Java classes from within the ScalaLab environment. The *JNI interface* and the *CUDA acceleration module* aim to provide very fast parallel operations based on GPU computing. The role of the other modules of the figure should be evident. As can be seen, ScalaLab is an integrated system with many classes for user interface tasks.

*Corresponding author: Stergios Papadimitriou, Department of Computer and Informatics Engineering, Technological Educational Institute of Eastern Macedonia & Thrace, 65404 Kavala, Greece. E-mail: sterg@teikav.edu.gr.

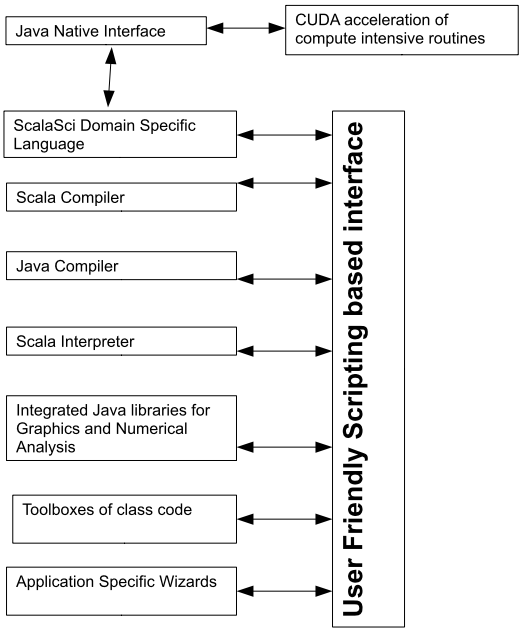

Fig. 1. The architecture of the main software components of ScalaLab.

The emphasis of this work is the important subject of interfacing Java matrix libraries in ScalaLab. These libraries are integrated within the core of ScalaLab. The aim is to provide an easy to use interface to these scientific libraries, without compromising their speed and effectiveness. Also, their functionality is retained and can be used from within ScalaLab with the native interface of each library. Furthermore, important computational tasks, e.g. FFT, can be performed within ScalaLab much faster with the new GPU acceleration feature (https://developer.nvidia.com/cuda-gpus).

ScalaLab is one of the few (if not the only) systems that integrates support for different matrix libraries. This multi-library support has these advantages:

(a) allows each user to work with the preferred library;
(b) permits comparison of results (both accuracy and performance) using the corresponding different libraries. Performances depend both on the storage format of matrices and the algorithms that each library uses;
(c) allows a "mixed mode" programming style, i.e. programming both with high-level constructs and with the native interface of each library. Routines from different libraries can also be combined for improved functionality;
(d) different libraries have their own strengths and weakness and which one is "better" depends on what each application requires.

The paper proceeds as follows. Section 2 describes the architecture for interfacing Java libraries in ScalaLab. Section 3 describes in some more detail the mechanism of implicit conversions that realizes high-level constructs without performance loss. Section 4 illustrates some basic techniques that are used to construct elegant high-level syntax, exploiting the superb facilities that the Scala language offers. Section 5 presents some examples of Java libraries used in ScalaLab. Section 6 introduces mixed mode programming, an effective practical framework for implementing numerical procedures. Section 7 presents a lot of performance benchmarks. Section 8 presents and discusses related work. Finally, Section 9 concludes the paper.

2. Interfacing libraries in ScalaLab

This section describes the main features of Scala that facilitate significantly the work with Java Scientific Libraries. The general architecture of interfacing Java libraries with ScalaLab is illustrated with Fig. 2.

For each library, two are the important interfacing classes: The *Wrapper Scala Class* (*WSC*) and the *Scala Object for Static Math Operations* (*SOSMO*). The *Wrapper Scala Class* (*WSC*) aims to provide a simpler interface to the more essential functionality of the Java library, e.g. for matrices A and B, we can add them simply as $A + B$, instead of the cumbersome $A.plus(B)$. For example, some wrapper Scala classes are the class *Matrix* for one-indexed matrices based on the NUMAL library [10], class *Mat* a zero-indexed matrix based on Scala implementations that borrow functionality from the JAMA Java (http://math.nist.gov/javanumerics/jama/) package and the *EJML.Mat* class based on the EJML library (https://code.google.com/p/efficient-java-matrix-library/).

The *Scala Object for Static Math Operations* (*SOSMOs*) provides overloaded versions of the basic routines for our new types. For example, it allows to use sin(B) where B can be an object of our *Mat* Scala class or a simple double number, or a Java 2-D array, etc. Each SOSMO object implements a large set of coherent mathematical operations. The rationale behind these objects is to facilitate the switching of the Scala interpreter to a different set of libraries. The top

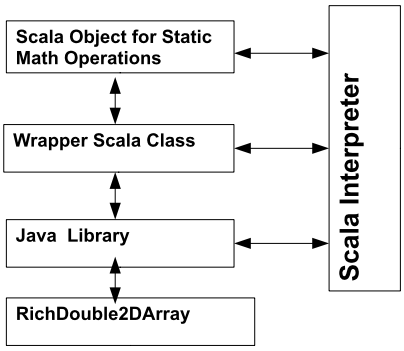

Fig. 2. The general architecture of interfacing Java libraries in ScalaLab.

level mathematical functions for the zero-indexed matrices e.g. *rand*0(*int* n, *int* m), *ones*0(*int* n) etc., should return the matrix type representation of the currently utilized library. A matrix object denoted e.g. *Mat* can refer to different matrices depending on the library. The "switching" of libraries is performed by creating a different fresh Scala interpreter that imports the corresponding libraries with the aid of the specially designed SOSMOs objects. For example, there exists a Scala object *StaticMathsJAMA* that performs important initializations for the JAMA library and a *StaticMathsEJML* for the Efficient Java Matrix Library one. The utilization of the JAMA library is accomplished by creating a Scala Interpreter that imports the *StaticMathsJAMA* object while for the EJML the *StaticMathsEJML* is imported.

The *RichDouble2DArray* is the "super" Matrix class of ScalaLab. It implements mathematical routines exploiting the best of different Java libraries. Also, it is available independently of the particular utilized library. By convention, utility routines that do not end in 0 or 1, return *RichDouble2DArray* objects. For example, *rand*(), *zeros*(), *ones*() etc., all construct *RichDouble2DArray* objects. Furthermore, the extensibility of *RichDouble2DArray* is exploited with implicit conversions [12] in order to provide for example its rich functionality to standard 2-D Java/Scala arrays. Also, a new feature the GPU acceleration, is implemented for many compute intensive routines of the *RichDouble2DArray*. This feature is based on the NVIDIA's CUDA framework (https://developer.nvidia.com/category/zone/cuda-zone).

The *Java library* module in Fig. 2 corresponds to the Java code of the library that performs the main numerical chores. We should note that the Scala interpreter can also use the native Java interface of each library.

There are many WSC classes, one for each supported library (e.g. JAMA, EJML, MTJ, Apache Common Maths). ScalaSci factors out the common patterns for all operations on zero-indexed matrices with the trait *scalaSciMatrix*. This factoring serves to avoid duplication of code and the corresponding inconsistency problems. The *scalaSciMatrix* trait leaves abstract only two methods:

- the *apply*(*i: Int, j: Int*)*: Double*, that returns the value of the corresponding matrix element. The Scala Compiler allows to call the *apply*() and *update*() functions with a MATLAB like syntax, e.g. the indexing operation $A(i, j)$ calls A.*apply*(i, j) and the update operation $A(i, j) = x$ is mapped to A.*update*(i, j, x).
- the *update*(*i: Int, j: Int, x: Double*), that assigns the value x to the corresponding matrix element.

Clearly, the implementation of these methods depends on the *storage format* of each matrix type.

Many other matrix methods are implemented in terms of the above two by the *scalaSciMatrix* trait. Therefore they are kept general and independent of the particular library. For example, we provide a row select operation like MATLAB's $A(i, :)$, by implementing an *apply* Scala method as in Listing 1.

At the code of Listing 1, a Scala object called *MatrixFactory* is used to construct the various matrix types. The *this* parameter at the call *scalaSci.MatrixFactory*(*this, rowNum, colNum*) corresponds to the particular matrix type where the *scalaSciMatrix* trait is mixed-in. Therefore, this object constructs a Matrix of the appropriate type, i.e. of the same type as the matrix that we extract its row.

The call involving *this* cannot be resolved statically by the Scala compiler, since the type of *this* is unknown at compile time. Thus these calls are dynamically binded according to the type of the object where the trait is mixed in. Therefore, this modular design with the factorization of the common code patterns in traits has a slight performance penalty. Specifically, the implementation of the access/update operations is only about 10–20% slower than the corresponding direct operations in the *RichDouble2DArray* class, that wraps two-dimensional Java double arrays and uses direct access/update operations for efficiency.

3. Implicit conversions

The mechanism of implicit conversions is central to the flexibility that Scala permits in customizing syntax, without compromising run-time efficiency. Sup-

Listing 1
Indexing with the apply methods

```
// extracts a specific row, take all columns, e.g. m(2, ::) corresponds to MATLAB's m(2, :)
  def apply(row: Int, allColsSymbol: ::.type): specificMatrix = {
    var colStart = 0; var colEnd = Ncols - 1;   // all columns
// dimensions for the extracted matrix (i.e. single row, all columns)
  var rowNum = 1; var colNum = colEnd - colStart + 1;
  var subMatr = scalaSci.MatrixFactory(this, rowNum, colNum)   // create a Matrix to keep the extracted range
// fill the created matrix with values
  var ccol = colStart
    while (ccol <= colEnd) {
        subMatr(0, ccol) = this(row, ccol)   // a dynamically binded call
        ccol += 1
        }
    subMatr.asInstanceOf[specificMatrix]
    }
```

pose that the statement $d+M$ is for evaluation where d is a double and M is a matrix object (i.e. of *Mat* class).

When the compiler detects an operator '+' on a Double object d that adds a Mat object M, i.e. $d+M$, it has a problem since this constitutes a type error. There is no method defined on the predefined Double type that adds to it a Mat object (and there cannot be one since Mat is a user library defined type). Similar is the situation when a Mat is added to a double array. Dynamic languages as Groovy [9], can easily overcome this obstacle by appending methods to the MetaClass of the Double or Double[] type. But when we do not want to sacrifice the merits of static typing other solutions should be searched.

Implicit conversions [12,20,21] provide efficient solutions in such cases in Scala. When an operation is not defined for some types, the compiler instead of aborting, tries any available implicit conversions that can be applied in order to transform an invalid operation to a valid one. The goal is to transform the objects to types for which the operation is valid.

The mechanism of implicit conversions is of fundamental importance for the construction of high-level mathematical operators in ScalaLab. Here, we describe the design of the implicit conversions in ScalaLab around the *RichNumber*, *RichDouble1DArray* and *RichDouble2DArray* classes.

Consider for example the code below:

var a = *rand* 0(200, 300)

//create a 200 *by* 300 *Matrix*

var $a2 = 2 + a$

//performs the addition by implicitly converting 2

The design of the implicit conversions for that chunk of code is build around the *RichNumber* class. This class models an extended Number capable of accepting operations with all the relevant classes of ScalaLab, e.g., with *Mat*, *Matrix*, *EJML.Mat, MTJ.Mat* and generally whatever class we need to process.

At the example above, the number 2 is transformed by the Scala compiler to a *RichNumber* object, that defines an operation to add a Matrix. Therefore, the operation proceeds effectively with only the slight cost of creating one *RichNumber* wrapper class.

Similarly, the classes *RichDouble1DArray* and *RichDouble2DArray* wrap the Array[Double] and Array[Array[Double]] Scala classes, in order to allow convenient operations, e.g. 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. Therefore, for example, the following code becomes valid:

var a = *Ones*(9, 10)

//an Array[*Array*[*Double*]] *filled with* 1 *s*

var $b = a + 10$

//add the value 10 *to all the elements returning* b *as an Array*[*Array*[*Double*]]

var $c = b + a * 89.7$

//similarly using implicit conversions this computation proceeds normally

4. Designing high-level operators in ScalaLab

The class used as an example to illustrate the design of high-level operators of ScalaLab is the *scalaSci.EJML.Mat* class. This class (abbreviated *Mat*) class in ScalaLab wraps the EJML *SimpleMatrix* class, allowing us to perform high-level MATLAB-like operations.

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 and keeps immutable the receiver objects. For example, to multiply matrix F and x we call $y = F.mul(x)$. The result of the multiplication is returned in y and F is kept unaltered. However, the Java like method calls are still not much convenient, for example to implement $P = FDF' + Q$ we have to write $P = F.mult(P).mult(F.transpose()).plus(Q)$ instead of the much clearer:

$$P = F * P * F\sim + Q$$

that we attain in ScalaLab.

In Scala operators on objects are implemented as *method calls*, even for primitive objects like Integers. But the compiler is intelligent enough to generate fast code for mathematical expressions with speed similar to Java. Therefore, in ScalaLab infix operators are implemented as method calls, e.g. the statement *var* $b = a * 5$ corresponds to *var* $b = a.*(5)$. Scala makes easy to implement *prefix* operators for the identifiers $+, -, !, \sim$ with the *unary_* prepended to the operator character. Also, *postfix* operators are methods that take no arguments, when they are invoked without a dot or parenthesis. Thus, for example, we can declare a method $\sim$ at the Matrix class and perform Matrix transposition in this way, i.e. to write the transpose of A as $A\sim$.

An example illustrating Scala's flexibility is the implementation of the MATLAB's colon operator. ScalaLab supports the MATLAB colon operator, for creating vectors, e.g. we can write:

var $t = 0 :: 0.02 :: 45.9.$

This statement returns t as a *scalaSci.Vec* type. To implement such syntax we combine *implicit conversions* with the *token cells approach* [5,6]. We construct also two helper objects, the *MatlabRangeStart*, and the *MatlabRangeNext*. In Scala, methods with name ending with the ':' character are invoked on their right operand, passing in the left operand. Therefore, the example is evaluated as 45.9. :: (0.02. :: (0)). Since *Double* (i.e. 45.9) does not have a :: method, it is implicitly converted to our *MatlabRangeStart* object. The *MatlabRangeStart* object retrieves the receiver's value (i.e. 45.9) with its constructor and stores it as the ending value of the range. We should remind, that with the order that Scala evaluates methods with name ending with ':', the first object (i.e. *MatlabRangeStart*) has as its value the end of the range. Consequently, the *MatlabRangeStart* object has a method named :: that processes the increment parameter. Note that :: is a valid method name in Scala. This method creates a *MatlabRangeNext* object passing the calling object (i.e. the MatlabRangeStart) as a parameter. The :: method of the *MatlabRangeNext* has all the information (i.e. starting, increment and ending values) to construct and return the vector.

MATLAB-like indexing/assignment is implemented easily by defining overloaded versions of *apply*() that operate on vectors. For example, at the sentence $M(2 :: 4 :: 20, 3 :: 2 :: 100)$, $M.apply(2 :: 4 :: 20, 3 :: 2 :: 100)$ is called. The implicit conversions mechanism, converts parameters to vectors, then apply extracts from the vectors the starting, increment and ending values, converting the call to the familiar $M.apply(2, 4, 20, 3, 3, 100)$.

Below we describe some key characteristics of important libraries of the ScalaLab core.

5. Example Java libraries of ScalaLab

Although there are many capable libraries, we describe only three as characteristic examples. The first one, the *EJML* is a Java library for many Linear Algebra tasks designed with efficiency in mind (i.e. both the algorithms and the matrix storage schemes are designed to be fast). The *NUMAL* library and the *Numerical Recipes* are general purpose Numerical Libraries. They offer wide support for many diverse numerical analysis tasks. Finally, *MTJ* is an object oriented wrapper to some essential functionality (actually a small part) of the famous *LAPACK* package.

5.1. The EJML library

The Efficient Java Matrix Library (EJML) is a linear algebra library for manipulating dense matrices. Its design goals are: (1) to be as computationally efficient

as possible for both small and large matrices, (2) to be accessible to both novices and experts and (3) to present three different interfaces: (a) the *SimpleMatrix*, (b) the *Operator* interface and (c) the *Algorithm* interface. These interfaces are ordered in increasing sophistication and run efficiency but also in decreasing simplicity (e.g. the Algorithm interface is the most efficient but also the most complicated).

These goals are accomplished by dynamically selecting the best algorithms to use at runtime and by designing a clean API. EJML is free, written all in Java, and can be obtained from http://code.google.com/p/efficient-java-matrix-library/. The documentation provided within the EJML sources is of superb quality with references to books on which the implementation of the algorithms is based. Also, the project is well documented with many examples for nontrivial applications, e.g. Kalman filters.

The EJML library stores matrices as one dimensional Java double array and in row major format, i.e. first the zero row of the matrix, then the second etc. The *CommonOps* class of EJML works by not overwritting the operands but instead it creates new objects for storing the results. This encourages a functional style of programming. The EJML is designed to facilitate the user, i.e. for a square matrix A of dimension *NXN* at the equation $Ax = b$, the exact solution is sought while for overdetermined systems (i.e. more equations than unknowns) the least squares solution is computed.

The EJML library provides an extensive set of functionality implemented with the efficiency goal in mind. For further efficiency, it implements many block algorithms and has the capability of switching automatically to block based processing, if for a particular matrix size it is faster.

In ScalaLab we can utilize the basic algorithms even more easily with high-level mathematical notation and with a MATLAB-like interface. The *SimpleMatrix* class of the Efficient Java Matrix Library implements mathematical operations in an object oriented way and keeps immutable the receiver objects. For example, to multiply matrix F and x we call $y = F.mul(x)$. However, the Java like method calls clearly are not convenient, for example to implement $P = F \cdot P \cdot F' + Q$ we have to write $P = F.mult(P).mult(F.transpose()).plus(Q)$ instead of the much clearer:

$$P = F * P * F\sim + Q$$

that we attain in ScalaLab. The *scalaSci.EJML.Mat* (abbreviated *Mat*) class in ScalaLab wraps the EJML *SimpleMatrix* class, allowing us to perform high-level MATLAB-like operations. In addition, we can exploit all the native potential of the EJML, as with any other library. Section 6 describes this topic.

5.2. The NUMAL and the Numerical Recipes libraries

Another basic library is the NUMAL Java library described in [10]. NUMAL has a lot of routines covering a wide range of numerical analysis tasks. NUMAL uses one-indexing of arrays as MATLAB and Fortran also do. The *Matrix* ScalaLab class is a one-indexed class designed to facilitate the ScalaLab user in accessing NUMAL functionality. The routines of that library operate on the Array[Array[Double]] type and they are imported by default in the Scala Interpreter. Also, the *DoubleDoubleArr* Scala object aims to implement an additional simpler interface to these routines. Additionally, an implicit conversion from Matrix to Array[Array[Double]] allows the NUMAL machinery to be used with the Matrix ScalaLab type. Although NUMAL is not object oriented, it is practically an effective framework for scientific computing, since it covers many engineering tasks and has good documentation (see [10]).

Similarly, the *Numerical Recipes* [19] is a library that covers a wide range of Numerical Analysis problems. The book [19] is an excellent tutorial to the internals of the routines, and therefore this library has a great tutorial value. The routines are generally fast and well designed and can be used from ScalaLab with their native Java interface. They operate on plain double Java arrays and therefore it is trivial to combine the functionality of Numerical Recipes with many other libraries.

5.3. The Matrix Toolkit for Java (MTJ) library

The Matrix Toolkit for Java (MTJ) is an open source Java matrix library (http://code.google.com/p/matrix-toolkits-java/) that provides extensive numerical procedures for general dense matrices, for various matrix categories (e.g. various band matrix forms), for block matrices and for sparse matrices. Most of the functionality of MTJ is built upon the powerful Java LAPACK (JLAPACK) package which is a Java translation of the famous LAPACK package [2]. The Netlib API obtained from the open-source project *netlib-java* (http://code.google.com/p/netlib-java/) provides a low-level interface to the Java LAPACK functionality. In turn, the Matrix Toolkits for Java (MTJ) project pro-

Listing 2
Eigenvalue decomposition using MTJ

```
// compute the eigenvalue decomposition of general matrix Mat
def eig(m: Mat) = {
/* compute the eigenvalue decomposition by calling a convenience method for computing the complete eigenvalue decomposition of the given matrix */
/* allocate an EVD object. This EVD object in turn allocates all the necessary space to perform the eigendecomposition, and to keep the results, i.e. the real and imaginary parts of the eigenvalues and the left and right eigenvectors
The method factorize() performs the eigendecomposition using JLAPACK. */
var evdObj = no.uib.cipr.matrix.EVD.factorize(m.getDM)  // getDM returns the MTJ matrix
                                                        // representation
(evdObj.getRealEigenvalues(), evdObj.getImaginaryEigenvalues(),
  new Mat(evdObj.getLeftEigenvectors()), new Mat(evdObj.getRightEigenvectors()))
}
```

vides a higher level API and is suitable for programmers who do not specifically require a low level Netlib API. We covered in detail the interfacing of MTJ and JLAPACK in ScalaLab in [14].

LAPACK is powerful but is difficult to use. The interfaces of their routines are Fortran like with long parameter lists, that make programming cumbersome and error prone.

With MTJ some part of the LAPACK functionality can be exploited much more easily. It uses an object-oriented design for its Matrix classes. For example, the *AbstractMatrix* is one of its basic base classes. A few methods of the AbstractMatrix through an *UnsupportedOperationException* and should be overridden by a subclass: *get*(*int, int*), *set*(*int, int, double*), *copy*() and all the direct solution methods. The concrete implementation of these methods is dependent on the matrix format that LAPACK uses for that particular matrix type, e.g. a tridiagonal band matrix, an upper-triangular etc. For the rest of the methods, the library provides simple default implementations using a matrix iterator.

Operations as the eigenvalue decomposition are kept with an object-oriented wrapping. The class EVD for example is used to compute eigenvalue decompositions of MTJ Dense Matrices by calling appropriately the powerful and reliable routines of the LAPACK library. After performing the eigendecomposition the user can conveniently acquire the results from the EVD object by calling appropriate methods, e.g. *getLeftEigenvectors*(), *getRightEigenvectors*(), *getRealEigenvalues*(), *getImaginaryEigenvalues*() etc.

ScalaLab constructs an additional layer in order to provide even more user friendly operations than MTJ. For example, the routine *eig*(*m: Mat*) performs the eigendecomposition of the MTJ Mat class m. This is achieved by factorizing first the MTJ matrix representation of the data, performing the eigendecomposition using JLAPACK's functionality and then preparing the results with a convenient Scala tuple for output, see Listing 2.

JLAPACK is powerful and provides advanced algorithms that generally perform well, even if the Java translation of the Fortran LAPACK code was obtained automatically. MTJ facilitates the utilization of the most important LAPACK functionality. However, with ScalaLab an expert programmer can exploit the full JLAPACK functionality with a scriptable way of work.

Finally, the JBLAS library (http://jblas.org/) offers a very usable interface to native BLAS and LAPACK routines. These routines, wrapped by the JBLAS class, are very easy to use from within ScalaLab. The relevant native libraries are installed automatically with the *jblas.jar* file. Therefore, using native routines from ScalaLab requires nothing special from the user. JBLAS routines are very fast, about 4 to 10 times faster than corresponding Java routines.

6. Exploiting and glueing the functionality of native libraries

ScalaSci implements many high-level Scala classes, that wrap Java classes of many scientific libraries. However, although it is more convenient to work with those Scala classes, they expose only the most important functionality of the lower-level classes. This section presents how we can adopt a *mixed mode programming style*, in order to exploit both the lower-level functionality and to have the convenience of Scala based operations where it is applicable.

We call that style as "mixed mode" since it consists of both ScalaSci code and library dependent

code patterns. Clearly, the engineer that uses the later type of code, should be familiar with the relevant library. Mixed mode programming is supported by four methods of the *scalaSciMatrix* trait: (a) *getNativeMatrixRef*(), (b) *matFromNative*(), (c) *FromDoubleArray*(), (d) *ToDoubleArray*(). Since all ScalaSci Matrix classes implement that trait they have this functionality.

The *getNativeMatrixRef*() method returns a reference to the library dependent class that implements native operations. This allows the ScalaLab programmer to combine all the rest functionality, with the existing native operations provided by the Java library. In this way the full potential of the underlying Java class can be utilized. The definition of *getNativeMatrixRef* as an abstract method of the ScalaSciMatrix trait is:

```
def getNativeMatrixRef: AnyRef
```

For example, this abstract method is implemented for the EJML ScalaSci classes as:

```
def getNativeMatrixRef() = sm
  //the scalaSci.EJML.EJMLMat wraps
    an EJML SimpleMatrix
```

Another important routine, the *matFromNative*, converts from the lower level representation to the corresponding higher level scalaSci matrix. Therefore, the *getNativeMatrixRef* can be used to take a reference to the lower level representation, transform it using routines of the native library and then convert back to the higher level scalaSci matrix using *matFromNative*.

Also, very useful for glueing the functionality of the different matrix libraries, are the routines *toDoubleArray*() that converts the contents of any matrix type to a simple two-dimensional Java array, and *fromDoubleArray*() that performs the opposite operation.

The *matFromNative* can be called using the scalaSci matrix reference on which *getNativeMatrixRef* is called, e.g.

```
var x = new scalaSci.Mat(4, 5)
  //create a ScalaSci matrix
var xv = x.getNativeMatrixRef
  //take its internal representation
xv(0)(0) = 200
  //change the internal representation,
    we can have here any operations
  //on the native representation
var xrecons = x.matFromNative(xv)
  //a new ScalaSci matrix with the changed
    data
```

7. Discussion on performance

Attempting to provide an open-source alternative to MATLAB for the Java Virtual Machine we initially implemented jLab [13,15,17], that is based upon an interpreter that executed MATLAB like scripts. However, the performance was very poor and the execution of reasonable loops formidable.

Compiled scripting was the solution to overcome these severe performance problems. Initially we used the Groovy [9] dynamically typed language at the context of jLab. Groovy was not quite efficient, although today as it is used with the *jlabgroovy* project (http://code.google.com/p/jlabgroovy/) has improved a lot in performance. With Groovy 2.0, the utilization of the new *invoke dynamic* instruction of the Java Virtual Machine and the possibility of statically compiling chunks of code, the performance is comparable to that of Scala/Java.

The HotSpot Java Virtual Machine JIT (Just In Time) compiler generates dynamic optimizations, in other words it makes optimization decisions while the Java application is running. Generally, it produces high performing native code.

By tuning the proper parameters of the JVM we can obtain a significant speedup. For example by setting for 32-bit Windows the parameters controlling the minimum and maximum heap size to 500 and 1500 Mb respectively, i.e. $-\text{Xms} = 500$ and $-\text{Xmx} = 1500$, we obtain execution time for the multiplication of a 2500×2500 matrix to 23.26 s. By adjusting the minimum heap in order to be the same as the maximum, i.e. $-\text{Xms} = 1500$, this time is reduced to 15.678 s, i.e. a speedup of about 35.34%.

We design ScalaLab with performance as one of basic goals. Therefore we have avoided using generic implementation of matrices since they produce much slower code. Also, the class *RichDouble2DArray* aims to combine the fastest algorithms from many libraries. In this way, *RichDouble2DArray* is a "super" matrix class, that provides its methods to the ScalaLab user in-

dependently of which particular matrix library is used by the Scala interpreter.

At the design of ScalaSci we avoid parameterized arrays, i.e. arrays of type Array[T] where T is any Java primitive type (i.e. int, long, float, double, byte, short, char or Object). These types require the Scala compiler to insert a sequence of type tests that determine the actual array type. As a consequence accesses to generic arrays are at least three to four times slower than accesses to primitive or object arrays.

Java conforms fully to IEEE standard [14] that establishes a precise interface for floating point computations. The double type performs sufficiently accurate for most applications and in today's JVMs is not slower than the float type. Therefore, generally ScalaLab prefers to present an interface for double types. Also, both Java and Scala support *BigDecimal* arithmetic at the library level. These types however are much slower. ScalaLab also supports the reliable computation framework of [6]. That framework can produce confidence intervals for many types of computations. Although the "*SmartFloat*" arithmetic runs much slower than pure double arithmetic, it can also be very useful at the development stage, since we can gain valuable insight about the accuracy of our algorithms.

JBLAS (http://www.jblas.org/) is similar in many aspects with MTJ in that it provides a higher level interface to BLAS and LAPACK functions. The NativeBLAS class of JBLAS contains the native BLAS and LAPACK functions. Each Fortran function is mapped to a static method of this class. For each array argument, an additional parameter is introduced which gives the offset from the beginning of the passed array. In C, we can pass a different reference, from the beginning of an array, but in Java, we can only pass the reference to the start of the array.

Due to the way the JNI (Java Native Interface) is implemented, the arrays are first copied outside of the JVM before the function is called. This means that functions whose runtime is linear in the amount of memory usually not run faster just because we are using a native implementation. This holds true for most Level 1 BLAS routines (like vector addition) but also for most Level 2 BLAS routines (matrix–vector multiplications).

JBLAS routines that use the Native BLAS are the fastest routines in ScalaLab, with nearly the same speed as corresponding MATLAB built-in operations (e.g. for matrix multiplications).

We performed several benchmarking tests comparing ScalaLab with SciLab and MATLAB. All the tests were performed on an Intel Core™ 2 Quad CPU clocked at 2.4 GHz, with 4 GB of RAM. Also, we compare with GroovyLab (http://code.google.com/p/jlabgroovy/) a similar system based on the Groovy dynamically typed language for the JVM. A general conclusion is that ScalaLab is significantly (i.e. about 2 to 5 times about) faster than SciLab but not fast as MATLAB for the operations that the latter implements with optimized built-in code. However, ScalaLab scripts run also significantly faster than M-file scripts.

It is interesting to observe that MATLAB's performance in some built-in operations (e.g. matrix multiplication) is similar to the performance we obtained from ScalaLab using Native BLAS (using the JBLAS library, http://www.jblas.org/). We can assume that MATLAB also uses these fast native routines.

Table 1 compares some qualitative aspects of ScalaLab and some similar environments. Also, Table 2 is concentrated on quantitative results on some characteristic problems for scientific programming.

In order to access the efficiency of accessing the matrix structure we have used the following simple script, for which we list the code in MATLAB.

Array access benchmark in MATLAB

$N = 2000; M = 2000$
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
$tm = toc$

For that script ScalaLab clearly outperforms both MATLAB and SciLab. GroovyLab has similar speed when the option of 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 meta-object 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 bytecodes.

The FFT benchmark is performed in ScalaLab using implementations of FFT from various libraries.

Table 1
Qualitative aspects of ScalaLab and some similar environments

	ScalaLab	MATLAB	SciLab	GroovyLab
Speed	Very fast, execution speed depends on the Java Runtime, generally about 2 times faster than MATLAB 2012 at script code, but slower for routines implemented as built-in with MATLAB	Very fast, especially the build-in routines 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	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 7 JRE	There exist versions for each main platform, e.g. Windows, Linux, MacOS	There exist versions for each main platform, e.g. Windows, Linux, MacOS	Very portable, anywhere exists installed Java 7 JRE
Open-source	Yes	No	Yes	Yes
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 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
Scalability of the language	The Scala language is designed to be scalable and extensible	MATLAB is not designed to be extensible	SciLab is not designed to be extensible	The Groovy language as dynamic is extensible
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

Of these libraries the Oregon DSP library obtains the fastest speed. The second and close in performance is the JTransforms (https://sites.google.com/site/piotrwendykier/software/jtransforms). Since JTransforms is multithreaded, it can logically gain superiority with better machines (e.g. having 8 or 32 cores, instead of only 4). Also, as can be seen, the rather tutorial FFT implementation of the classic Numerical Recipes book [19] obtains adequate performance. Surprising enough is that the Oregon DSP and JTransforms FFT routines are nearly as fast as the optimized built-in FFT of MATLAB.

We tested also other types of problems such as the eigenvalue decomposition, singular value decomposition, solution of overdetermined systems etc. The general conclusion is that ScalaLab is faster than SciLab 5.21 by about 3 to 5 times but slower than MATLAB 7.1 by about 2 to 3 times. It is evident also that routines of JLAPACK for special matrix categories run orders of magnitude faster than routines for general matrices, e.g. for a 1500×1500 band matrix with 2 bands above and 3 bands below the main diagonal, the JLAPACK's SVD routines runs about 250 times faster than for a general 1500×1500 matrix.

In order to test the JVM performance vs native code performance, an implementation of SVD in C is used (see http://code.google.com/p/scalalab/wiki/ScalaLabVsNativeC). We used both the Microsoft's *cl* compiler of Visual Studio on Windows 8 64-bit, and the *gcc* compiler running on Linux 64-bit. ScalaLab is based on the Java runtime version: 1.7.0_25 and Scala 2.11 M4. Clearly, ScalaLab is faster than unoptimized

Table 2
Results of some basic benchmarks

	ScalaLab (s)	SciLab 5.21 (s)	MATLAB 7.1 (s)	GroovyLab (s)
Matrix multiplication with matrix sizes: (2000, 2500) × (2500, 3000)	36.5 with Java, 14.2 with Native BLAS	61.8	13.05	The same as ScalaLab
LU				
1000	0.67	3.13	0.36	0.7
1500	2.41	3.82	1.18	2.58
2000	5.6	6.42	2.72	5.8
inv				
1000	2.7	12.97	1.3	3.1
1500	7.8	13.14	4.5	8.2
2000	9.31	19.07	5.9	10.1
QR				
1000	2.5	4.3	1.2	2.7
1500	11.3	9.96	4.26	12.4
2000	29.09	19.69	9.89	30.2
Matrix access scripting benchmark	0.03	32.16	10.58	0.031 static compilation, 0.156 with primitive ops, 0.211 with invoke dynamic
FFT 100 ffts of 16,384 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*: 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

Note: These results are obtained by averaging 5 executions.

Table 3
SVD performance: Java vs native C code

Matrix size	Optimized C (*gcc*, similar is for *cl*)	ScalaLab	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

C and close even to optimized C code. Table 3 shows some results.

For obtaining the SVD code from the third edition of the Numerical Recipes book [19] is used. The same code is implemented both in C++ and in Java, therefore direct comparison can be performed. The results presented in Table 3 are obtained by averaging 5 runs. However, the execution time is stable, i.e. we have not observed significant deviations.

8. Related work

This section compares the ScalaLab environment with some other related approaches.

Similar to ScalaLab is the GroovyLab system, that is developed also as an open source project (https://code.google.com/p/jlabgroovy/). It uses Groovy instead of Scala for scripting. There are many similarities to the user interface design between the two systems, but also many essential differences concerning both execution speed and syntax constructs. We plan to present an extensive comparison of the two systems with another article. Since both systems offer similar functionalities,

which fits better depends to a large degree upon which language, Groovy or Scala, the programmer is more familiar with.

Similar to many aspects with ScalaSci is the *Breeze* system (https://github.com/scalanlp/breeze). It is developed in pure Scala and presents also elegant MATLAB like systax as ScalaSci, for a lot of mathematical problems. However, Breeze is a library and not an integrated MATLAB-like environment as ScalaLab. In addition, the functionality of Breeze can be made available to the ScalaLab user, if we install Breeze as a ScalaLab toolbox.

Similarly, Spire (https://github.com/non/spire) is an elegant Scala library, that provides a rather limited set of classes. It focus on efficiency using advanced facilities of Scala, as Scala macros. In contrast, ScalaLab obtains efficiency by avoiding parameterized types using carefully designed matrix structures and Java multithreading where is appropriate. Also, ScalaLab is now developed with integrated CUDA support (https://developer.nvidia.com/category/zone/cuda-zone). CUDA makes possible to obtain extremely fast computations using NVIDIA graphics cards. For example, on a GeForce GTX 650 Ti BOOST card, a 3000×3000 matrix multiplication is about 110 times faster than Java.

Finally, similar systems is the *NumPy* [18] based on Python and the *jHepWork* [3] based on Jython. These systems offer similar to ScalaLab set of tools, although with a different syntax based on Python. In addition, Jython allows as Scala, to use any Java library. Since Scala is statically typed, it has a speed advantage compared to Jython. Also, it allows to exploit the functional programming style that is very powerful. However, which system fits better for some application, depends both to the peculiarities of the application and on the programmer's preferences.

9. Conclusions and future work

This work has presented some ways by which we can work more effectively with existing Java scientific software from within ScalaLab. We demonstrated that ScalaLab can integrate elegantly well-known Java numerical analysis libraries for basic tasks. These libraries are wrapped by Scala objects and their basic operations are presented to the user with a uniform MATLAB-like interface. Also, any specialized Java scientific library can be explored from within Scalalab much more effectively and conveniently.

Scala is ideal for our purpose: the ability to handle functions as first-class objects, the customizable syntax, the ability to overload operators, the speed of the language, the full Java interoperability, are some of its strengths. An extension of Scala with MATLAB-like constructs, called ScalaSci is the language of ScalaLab. ScalaSci is effective both for writing small scripts and for developing large production level applications.

ScalaLab emphasizes user friendliness and therefore develops facilities 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 concentrates on improving the interfaces of Java basic libraries and on incorporating smoothly more competent libraries (e.g. the COLT library for basic linear algebra). Also, we work on providing better on-line help and code-completion for these routines. These facilities are of outstanding importance and support significantly the utilization of these rather complicated libraries. Also, another important direction for work is parallel programming with the NVIDIA's CUDA framework (https://developer.nvidia.com/cuda-gpus). We started to work with CUDA interfaces to some important routines, and the results seem very promising. For example we obtain a speedup of more than 50 for multiplication of large matrices, and more than 30 for FFTs of large signals. We plan to report on the CUDA interface of ScalaLab in a future article.

References

[1] A. Aho, M.S. Lam, R. Sethi and J.D. Ullman, *Compilers, Principles, Techniques, & Tools*, 2nd edn, Addison-Wesley, 2007.

[2] E. Anderson, Z. Bai, C. Birschof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. Mckenney and D. Sorensen, *LAPACK Users' Guide*, 3rd edn, SIAM, 1999.

[3] S.V. Chekanov, *Scientific Data Analysis using Jython Scripting and Java*, Springer-Verlag, 2010.

[4] E. Darulova and V. Kuncak, Trustworthy numerical computation in Scala, *ACM OOPSLA'11*, October 22–27, Portland, OR, USA, 2011.

[5] G. Dubochet, On embedding domain-specific languages with user-friendly syntax, in: *Proceedings of the 1st Workshop on Domain Specific Program Development*, 2006, pp. 19–22.

[6] G. Dubochet, Embedded domain-specific languages using libraries and dynamic metaprogramming, PhD thesis, EPFL, Suise, 2011.

[7] K. Hinsen, The promises of functional programming, *Computing in Science & Eng.* **11**(4) (2009), 86–90.

[8] C. Horstmann and G. Cornell, *Core Java 2, Vol. I – Fundamentals, Vol. II – Advanced Techniques*, 8th edn, Sun Microsystems Press, 2008.

[9] D. Konig, A. Glover, P. King, G. Laforge and J. Skeet, *Groovy in Action*, Manning Publications, 2007.

[10] H.T. Lau, *A Numerical Library in Java for Scientists and Engineers*, Chapman & Hall/CRC, 2003.

[11] K. Laufer and G.K. Thiruvathukal, The promises of typed, pure and lazy functional programming, Part II, *Computing in Science & Eng.* **11**(5) 68–75.

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

[13] S. Papadimitriou, Scientific programming with Java classes supported with a scripting interpreter, *IET Software* **1**(2) (2007), 48–56.

[14] S. Papadimitriou, S. Mavroudi, K. Theofilatos and S. Likothanasis, The software architecture for performing scientific computation with the JLAPACK libraries in ScalaLab, *Scientific Programming* **20** (2012/2013), 379–391.

[15] S. Papadimitriou and K. Terzidis, jLab: Integrating a scripting interpreter with Java technology for flexible and efficient scientific computation, *Computer Languages, Systems & Structures* **35** (2009), 217–240.

[16] S. Papadimitriou, K. Terzidis, S. Mavroudi and S. Likothanasis, ScalaLab: an effective scientific programming environment for the Java Platform based on the Scala object-functional language, *IEEE Computing in Science and Engineering (CISE)* **13**(5) (2011), 43–55.

[17] S. Papadimitriou, K. Terzidis, S. Mavroudi and S. Likothanasis, Scientific scripting for the Java platform with jLab, *IEEE Computing in Science and Engineering (CISE)* **11**(4) 2009, 50–60.

[18] F. Perez, B.E. Granger and J.D. Hunter, Python: An ecosystem for scientific computing, in: *IEEE Computing in Science & Engineering*, March/April 2011, pp. 13–21.

[19] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, *Numerical Recipes in C++, The Art of Scientific Computing*, 2nd edn, Cambridge Univ. Press, 2002.

[20] V. Subramaniam, *Programming Scala – Tackle Multicore Complexity on the Java Virtual Machine*, Pragmatic Bookself, 2009.

[21] D. Wampler and A. Payne, *Programming Scala*, O'Reily, 2009.

[22] T. Wurthinger, C. Wimmer and H. Mossenblock, Array bounds check elimination for the Java HotSpot client compiler, in: *PPPJ 2007*, September 5–7, ACM, Lisboa, Portugal, 2007.