Published in IET Software
Received on 21st October 2010
Revised on 20th May 2011
doi: 10.1049/iet-sen.2010.0135

ISSN 1751-8806

Exploiting Java scientific libraries with the Scala language within the ScalaLab environment

S. Papadimitriou[1] *K. Terzidis*[1] *S. Mavroudi*[2] *S. Likothanassis*[2]

[1]*Department of Information Management, Technological Educational Institute of Kavala, 65404 Kavala, Greece*
[2]*Pattern Recognition Laboratory, Department of Computer Engineering and Informatics, School of Engineering, University of Patras, Rion, Patras 26500, Greece*
E-mail: sterg@teikav.edu.gr

Abstract: Since Java is one of the most popular languages in the academic and research community a lot of robust and effective scientific libraries have been developed. However, the utilisation of these libraries is very awkward especially for the average scientist that does not expertise in software development. The study presents the framework that has constructed for the utilisation of Java scientific libraries within the ScalaLab environment. The flexibility and extensibility of the Scala language allows the implementation of simple, coherent and efficient Matlab-like interfaces to those libraries. Moreover, other specialised Java libraries can be exploited much more easily and productively from within ScalaLab with the toolbox import mechanism that this work describes. Additionally, the system offers facilities such as on-line help, code completion, graphical control of the class-path and a specialised text editor with code colouring facilities that greatly facilitate the development of scientific software.

1 Introduction

Numerical computation applications benefit from 'interactive systems for matrix computation', which facilitate the rapid scientific experimentation. With these scripting systems substantial analysis can be performed by entering stepwise commands and thereby obtaining results. Experimentation is encouraged and the tedious 'compile-link-execute' cycle of standard programming languages is eliminated. Well-known examples of such systems include commercial products such as Matlab, Maple and Mathematica, and open source packages like Scilab and Octave.

Recently, we introduced two open source mathematical programming environments for the Java virtual machine (JVM), jLab [1] and ScalaLab [2]. Although these systems present a Matlab-like style of working, they differ from the forementioned in that they compile the scripts for the JVM. They utilise and expand modern languages for the JVM, jLab builds upon the dynamic Groovy language [3] while ScalaLab exploits the powerful Scala object-functional one [4].

Interactive scientific programming environments have gained much popularity mainly for their simplicity and the great speed improvements with just in time (JIT) compilation techniques. It is interesting to observe that the recent versions of Matlab have also gained impressive speed improvements. However, the bulk of numerical analysis software is in compiled languages, mainly in Fortran, C/C++ [5] and Java. The later language although is not designed specially for numerical computation has become very popular because of its portability, reliability, extensibility, simplicity and the advances in JIT compilation. We motivated our present work in order to develop a user friendly framework for the effective utilisation of these Java-based scientific libraries within ScalaLab.

The ScalaLab environment [2] builds upon the Scala language system. ScalaLab is an open-source project and can be obtained from http://code.google.com/p/scalalab/. The general high-level architecture of ScalaLab is depicted in Fig. 1. An essential component of ScalaLab, on which interactivity and user friendly operation is built upon, is the Scala interpreter. The overall approach of the Scala's interpreter is based on compiling initially the requested code. Afterwards, a Java classloader and Java reflection are used to run the code and access its results.

The general architecture of ScalaLab and its programming model is described in [2]. In this article, we concentrate on two specific directions not addressed in [2]. The first one, is the incorporation of Java numerical libraries within the core of ScalaLab. The aim is to provide an easy to use interface to these scientific libraries, without compromising their effectiveness. Convenient syntax features such as high-level mathematical operators are implemented by exploiting the rich support that Scala provides. Many features of Java's Swing [6] assist the work of the user, for example by providing extensive help and superb display functionality. Moreover, a set of basic functions is kept consistent and independent of the utilised library. Also we seek for a Matlab-like syntax for these functions in order to facilitate the user.

The second direction is the utilisation of external Java libraries of scientific code for toolboxes. The system assists

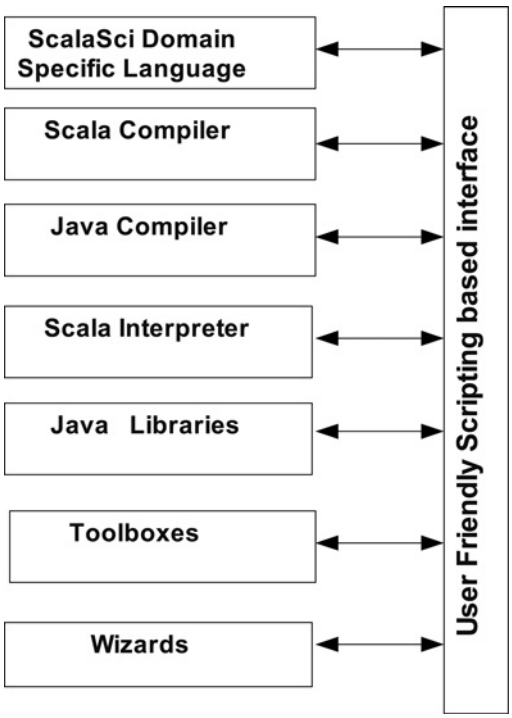

Fig. 1 *Architecture of the main software components of ScalaLab*

the user by providing insight about the contents of the toolboxes using the interrogation potential of the Java reflection API.

The paper proceeds as follows: Section 2 describes the way that we can wrap Java scientific libraries with Scala and make them more conveniently accessible, Section 3 presents a particular example that utilises a popular open-source Java numerical library, the efficient Java matrix library (EJML) (http://code.google.com/p/efficient-java-matrix-library/). Section 4 describes the utilisation of specialised Java code with the toolbox import mechanism. Section 5 presents an example of toolbox application using the FastICA Java toolbox. Section 6 evaluates our approach and compares it with other related ones. Finally, Section 7 concludes the paper and oulines some directions for future work.

2 Decorating Java scientific libraries with Scala

In this section, we describe the main features of the Scala language that we have explored in order to facilitate significantly the work with Java scientific libraries.

The general, architecture of interfacing Java libraries is illustrated with Fig. 2. The 'Wrapper Scala class' aims to provide a simpler interface to the more essential functionality of the Java library, for example, for matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, we can add them simply as $\boldsymbol{A}+\boldsymbol{B}$, instead of the cumbersome $\boldsymbol{A}$.plus($\boldsymbol{B}$). Also, it performs the useful task of transforming interfaces to a common pattern, for example, each Java matrix library has its own style of returning eigenvalues and eigenvectors. We have to adopt a single one (preferably Matlab-like) in order not to confuse the user.

The 'Scala Object for Static Math Operations' aims to provide overloaded versions of the basic routines for our new type. (e.g. to be able to use sin(A) where A is an object of our wrapper Scala class).

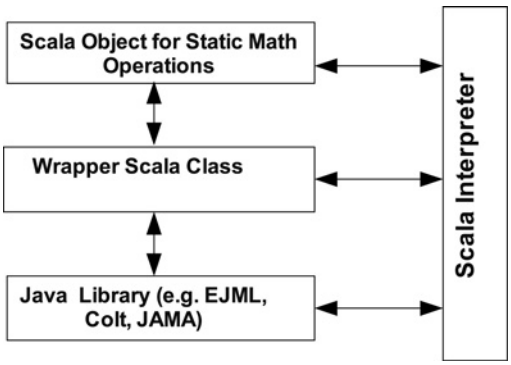

Fig. 2 *General architecture of interfacing Java libraries*

We should note that the Scala interpreter has access to these three modules and thus it can use even the native Java interface of the library.

Below we elaborate on the Scala's approaches of extending and decorating Java libraries by presenting a simple illustrative example for the implementation of a class that implements Matrices. This class is the *Mat* class.

The 'SimpleMatrix' class of the 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 $\boldsymbol{F}$ and $\boldsymbol{x}$ we call $y = \boldsymbol{F}.mul(\boldsymbol{x})$. However, the Java-like method calls are still not much convenient, for example to implement $\boldsymbol{P} = \boldsymbol{F}\ \ \boldsymbol{D}\ \ \boldsymbol{F}' + \boldsymbol{Q}$, we have to write $\boldsymbol{P} = \boldsymbol{F}.mult(\boldsymbol{P}).mult(\boldsymbol{F}.transpose()).plus(\boldsymbol{Q})$ instead of the much clearer:

$\boldsymbol{P} = \boldsymbol{F}^{*}\boldsymbol{P}^{*}\boldsymbol{F} \sim + \boldsymbol{Q}$ 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.

The default (primary) constructor performs global initialisation operations (since in Scala any auxiliary constructor must call finally the primary constructor). It keeps a reference to the SimpleMatrix object. This primary constructor is shown:

```
class Mat(smi: SimpleMatrix) = {
var sm = smi // keep a reference to the SimpleMatrix
// getters for size
def Nrows = smi.numRows()
def Ncols = smi.numCols()
def size = { ( Nrows, Ncols) }
def length = { Nrows }
```

It is interesting that Scala does not have operators. The flexible syntax of method calls and the acceptance of many special symbols as method names permits the implementation of methods that give the illusion of the built-in operators. Consider for example the operations

```
var a = 100
var b = a + 40
```

The operator '+' is actually a method call on the Integer object *a*, that is, we can write it more verbosely as:

var b = a .+ (40). The operator-like compact syntax is possible since Scala does not require the dot ('.') symbol for method calls and parenthesis are optional when a method has a single argument.

Therefore in Scala operations on objects are implemented as method calls, even for primitive objects like Integers.

doi: 10.1049/iet-sen.2010.0135

However, the compiler is intelligent enough to generate fast code for mathematical expressions with speed similar to Java.

Scala makes easy to implement 'prefix' operators for the identifiers +, −, !, ~ 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 ~ at the Matrix class and perform matrix transposition in this way, that is, to write the transpose of $\boldsymbol{A}$ as $\boldsymbol{A}\sim$.

Returning to our matrix 'Mat' class example, when the compiler detects an operator '+' on a Double object *d* that adds a Mat object M, that is, $d + \boldsymbol{M}$, 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 [3], 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 [4, 7, 8] 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.

For the 'Mat' class we define some implicit conversions that cope with the usual cases where a Mat needs to act upon a predefined object (e.g. a Double) appearing before it at the expression.

These implicits are for example as

```
implicit def DoubleToMat(x: Double) = scalaSci.StaticMaths.
fill0(1,1,x) // implicit conversion of a Double to Mat
```

At this example the implicit handles the case when the receiver is a Double. Suppose we have an expression $d + \boldsymbol{M}$ where *d* is a Double number and $\boldsymbol{M}$ is a matrix 'Mat' object. The operator '+' (method actually as we described), should add the Double number *d* with the corresponding elements of the Mat object $\boldsymbol{M}$. However, the Double type is of a predefined type (i.e. built-in) and cannot have any provisions for implementing operators (methods) that take arguments of user defined types. However, the Scala compiler does not give up! Instead of aborting with a compilation error, it tries to detect if the double array type is convertible to something that can be added with a Mat with the '+' operator. The compiler tries to unblock by exploring all the available implicit conversions within scope. Indeed, in our case, the 'DoubleToMat' conversion takes a Double and converts it to a Mat, with the method *fill0* defined at the *scalaSci.StaticMaths* package. This mechanism is powerful since it allows us to utilise existing code libraries according to our demands. Independently of the number of implicits the overhead of implicit processing at the generated code is absent since all the processing is done at compile time. The only overhead at run-time, is the overhead to perform the implicit conversion, which usually can be negligible.

Scala programmers can implement syntactically elegant indexing on any objects with the 'apply' method and assignment of values with the 'update' method. These simple things are of utmost importance for the conveniency of the interface. For the *Mat* class, obviously we want if *M* is a Mat to access its (i, j)th element as $\boldsymbol{M}(i, j)$. Thus we implement the apply method as

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

We note that the 'apply' method calls the corresponding routine 'get' of the EJML library. The Scala compiler supports flexible syntax for the apply and update methods, for example, we can call $\boldsymbol{M}(i, j)$ instead of *M.apply(i,j)* and write: $\boldsymbol{M}(i, j) = 9.8$ instead of *M.update(i, j, 9.8)*.

The corresponding 'update' operation implements assignment of elements and can be implemented as

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

The 'apply' method can be easily overloaded in order to extract a Mat subrange by implementing the method apply as

```
def apply( rowStart: Int, rowInc: Int, rowEnd: Int, colStart:
Int, colInc: Int, colEnd: Int) = {
  // the routine extracts and returns a Mat subrange with a
new Mat object
  ....
  }
```

The end result of this design is that the user can perform convenient operations on matrices, for example, $\boldsymbol{M}(2, k, m, 4, 2, N)$ to extract a range denoted in Matlab as $\boldsymbol{M}(2{:}k{:}m, 4{:}2{:}N)$.

Scientific programming environments demand for a global namespace of functions. Scala has no globally visible methods; every method must be contained in an object or a class. However, a global function namespace can be implemented easily with 'static imports'. Therefore by creating global objects we have the same convenience as if global methods existed. For example the plot() method is available since we import it from object scalaSci.plot.plot. Also, Scala offers the possibility to define 'apply()' methods for the companion objects of classes. These apply() methods offer the convenience to call them directly with the object name. In this case, we need to import in the global environment only the object and not the particular method.

3 Examples of interfaced Java scientific libraries

ScalaLab provides a novel feature: a uniform high-level interface to multiple Java numerical libraries that can be switched in order the user to investigate their relative benefits easily. Currently, we utilise the NUMAL library [9] with the class *Matrix* that offers one-indexed Matrices since the double [][] arrays in NUMAL are one-indexed. Although, C/C++ and Java programmers are familiar with zero-indexed arrays, Fortran and Matlab are widespread languages popular at the scientific community that use one-indexed arrays. Thus, this is another additional reason to keep a one-indexed Matrix class. Also, for the zero-indexed *Mat* class we can switch the library on which this class is based: the current options are the JAMA library (http://math.nist.gov/javanumerics/jama/), the EJML (http://code.

google.com/p/efficient-java-matrix-library/) and the Matrix Toolkit for Java (MTJ) (http://code.google.com/p/matrix-toolkits-java/).

This is accomplished by implementing similar interfaces with the corresponding Scala wrapper classes to these libraries. Also, we take care to make the library routines Matlab-like whenever possible, to facilitate users familiar with Matlab.

As we described (see Fig. 2), in order to interface Java scientific code to ScalaLab, a wrapper Scala class is created that provides high-level operations by defining operator-like methods, for example, '+' for addition. For example, the *EJML.Mat* Scala class provides a more convenient interface to the functionality of the 'SimpleMatrix' Java class of the EJML library. This class implements the user friendly uniform interface to the Java class. Thus the main benefits of this class are twofold:

(a) Providing operator-like methods, for example, '+', '*', etc.
(b) Designing similar methods for all interfaced libraries and, preferably one similar to Matlab, in order the user to have a familiar and uniform interface.

The top-level mathematical matrix functions, for example, *rand(int n, int m)*, *ones(int n)*, etc. should cope with the matrix-type representation appropriate to the currently utilised library. Also, a matrix object denoted for example, *Mat* can refer to different matrices depending on the library. The 'switching' of libraries is performed by initialising a different Scala interpreter that imports the corresponding libraries. For example, there exists a Scala object 'StaticMathsJAMA' that performs important initialisations for the JAMA library and a 'StaticMathsEJML' for the EJML one. The utilisation of the JAMA library is accomplished by creating a Scala Interpreter that imports the 'StaticMathsJAMA' object while for the EJML the 'StaticMathsEJML' one is imported. Currently, the ScalaLab user can switch different underlying Java libraries with a right-mouse click popup menu. We will improve the user interface in future ScalaLab versions, in order to allow the user to specify more conveniently the preferences on the Java scientific libraries that he (she) wants to have in the working configuration.

The EJML is a linear algebra library for manipulating dense matrices. Its design goals are: (i) to be as computationally efficient as possible for both small and large matrices, (ii) to be accessible to both novices and experts and (iii) 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 EJML library stores matrices as one-dimensional Java double array and in row major format, that is, 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, that is, for a square matrix $\boldsymbol{A}$ of dimension N × N at the equation $\boldsymbol{Ax} = \boldsymbol{b}$, the exact solution is seeked while for overdetermined systems (i.e. $N > M$) the least squares solution is computed.

The library provides an extensive set of functionality implemented with the efficiency goal in mind. In ScalaLab, we can utilise the basic algorithms even more easily and with a Matlab-like interface.

4 Toolboxes of scientific code

In the previous section, we investigated the integration of important numerical analysis libraries within the ScalaLab kernel. These libraries provide a set of useful general purpose functions, for example, rand, svd, eig, rank to the ScalaLab user. These functions although they seem hardwired to the system are implemented on top of Java numerical analysis libraries using Scala classes and objects to wrap their operation. However, for more specialised libraries of scientific code another mechanism is designed that utilises them dynamically without embedding any code within the ScalaLab core. An important advantage of Scalalab is its ability to cope easily and with a Matlab-like scripting with existing libraries of Java scientific code. ScalaLab exploits the reflection capabilities of Java in order to interrogate dynamically the Java libraries and to present graphically their class contents.

There are two methods by which the ScalaLab user can utilise toolboxes. The first method that is simpler but does not present an insight about the toolbox contents is to simply place the classes of the toolbox at the classpath of the Scala interpreter.

This is accomplished by the following steps:

1. Unziping the .jar file of the toolbox.
2. Updating the *ScalaClassPath* to include the root directory where the .jar file was extracted.

Alternatively, we can append the ScalaLab classpath with the .jar file of the toolbox without unzipping it. This choice has the advantage of avoiding the creation of many files on disk.

The second method of installing toolboxes is more convenient since it guides dynamically the user about the toolbox contents using the reflection potential of Java. The classes and public methods available from the toolbox are displayed graphically using the potential of the 'JTree' [6] Swing display component to display hierarchically with a tree representation, the information extracted with the Java reflection API (Fig. 3).

With this method we import the specified toolboxes in the 'Available Toolboxes' list with the 'Import toolboxes' button. Toolboxes can be removed from the 'Available Toolboxes' list with a right-mouse click.

An installed toolbox during one ScalaLab session, remains on the ScalaClassPath, for example, having installed weka.jar, weka.jar remains on the ScalaLabClassPath for next sessions.

Toolboxes can also be easily removed by:

1. Selecting the toolbox from the loaded toolboxes list.
2. Right-mouse click and selecting the remove option of the popup menu.

Toolbox classes are loaded into the JVM with an instance of 'JarClassLoader' class which itself extends the class 'ExtensionClassLoader'. The ExtensionClassLoader class extends the standard Java application 'ClassLoader' class. It keeps a list of additional paths at which it is able to search

doi: 10.1049/iet-sen.2010.0135

Fig. 3 *Classes of the Java Fast ICA toolbox retrieved with reflection and displayed graphically with a JTree*

for a class. Additional entries can be added to this list, in order to extend the classpath. The ExtensionClassLoader first delegates the class-loading task to the standard Java System Class Loader (or Application Class Loader). If the later fails to load the class then the ExtensionClassLoader searches to find the class in the specified paths. We should note that the ExtensionClassLoader favors predefined Java classes over user classes of the same name. This conforms to the usual Java security policy of avoiding to override system classes [6].

The JarClassLoader class offers special convienient methods for retrieving Java classes from .jar files. The JarClassLoader loads the classes by first creating a Scala Interpreter with an extended classpath that includes the toolbox .jar file. Also, an ExtensionClassLoader is created that also utilises the same extended classpath. This class loader has as its parent the classloader of the Scala interpreter.

We demonstrate below the ways that the ScalaLab environment facilitates the utilisation of Java scientific libraries. The Java library used as an example toolbox performs independent component analysis (ICA) using the FastICA algorithm.

5 FastICA toolbox

The FastICA algorithm exploits the notion of non-Gaussianity, which is a requirement of ICA [10].

A Gaussian random variable distinguishes itself from all other random variables by having the largest possible differential entropy. In particular, the information content of a Gaussian random variable is confined to second-order statistics, from which all higher-order statistics can be computed. To access the non-Gaussianity of a random variable, we postulate a measure that satisfies two properties:

(a) The measure is non-negative, assuming the limiting value of zero for a Gaussian random variable.

(b) For all other variables, the measure is greater than zero.

The concept of negentropy satisfies both of these properties.

Consider a random vector $\boldsymbol{X}$ that is known to be non-Gaussian. The 'negentropy' of $\boldsymbol{X}$ is formally defined by

$$N(\boldsymbol{X}) = H(\boldsymbol{X}_{\text{Gaussian}}) - H(\boldsymbol{X})$$

where $H(\boldsymbol{X})$ is the differential entropy of $\boldsymbol{X}$ and $H(\boldsymbol{X}_{\text{Gaussian}})$ is the differential entropy of a Gaussian random vector whose covariance matrix is equal to that of $\boldsymbol{X}$.

In information-theoretic terms, negentropy is an elegant measure of non-Gaussianity; however, is highly demanding from the computational point of view. Therefore simple approximations to negentropy are utilised by the FastICA algorithm. It is beyond our scope to describe the details. The interesting reader can consult the excellent theoretical presentation of [10].

The example below demonstrates the utilisation of the FastICA with ScalaLab. This Java library is open source and can be obtained from http://sourceforge.net/projects/fastica/. Although, the FastICA library can be used with any Java program, ScalaLab facilitates thinks significantly. The ScalaSci script that follows starts by constructing two synthetic sinusoidal signals and a Gaussian random signal (with the vrand() call). Then it mixes them by performing a Matrix multiplication. Subsequently, a FastICAConfig structure is configured in order to initialise properly parameters for the specific FastICA library. Also, a listener component is created to watch at the FastICA computation. For readers familiar with Java, the progressListener has the same semantics as the Java's Swing listeners, although with Scala's syntax. Finally, an activation function for the FastICA algorithm is chosen and the algorithm is performed. Fig. 4 displays the original signals before mixing at the left plot and those after performing ICA at the right plot. We observe that the ICA algorithm separates the components successfully (Fig. 5).

6 Comparative evaluation of our method

The traditional approach for scientific scripting environments is to build them using interpreted scripting approaches. This was the approach of the 'j-Script' interpreter [11, 12] and of open-source packages as SciLab and Octave. However, interpreted scripting usually cannot confront heavy computational loops, since it is much slower than compiled scripting.

Compiled scripting frameworks for Java operate by first compiling the code to class bytecodes. Then a 'Java classloader' is used together with a variable 'binding' scheme to execute the class and communicate its results.

Compiled Scripting apart from the speed advantage provides the potential to directly call Java class code. In contrast, for external class code to be called from the *j*-Script interpreter we have required for each function that needs to be called externally to implement an 'external' function interface that consisted of an 'evaluate()' function [11, 12]. The interfacing of existing Java libraries required a significant work to implement the appropriate interface functions. Also, specialised toolboxes of Java code practically cannot be utilised, since their classes are not directly accessible and subtle interfaces are required.

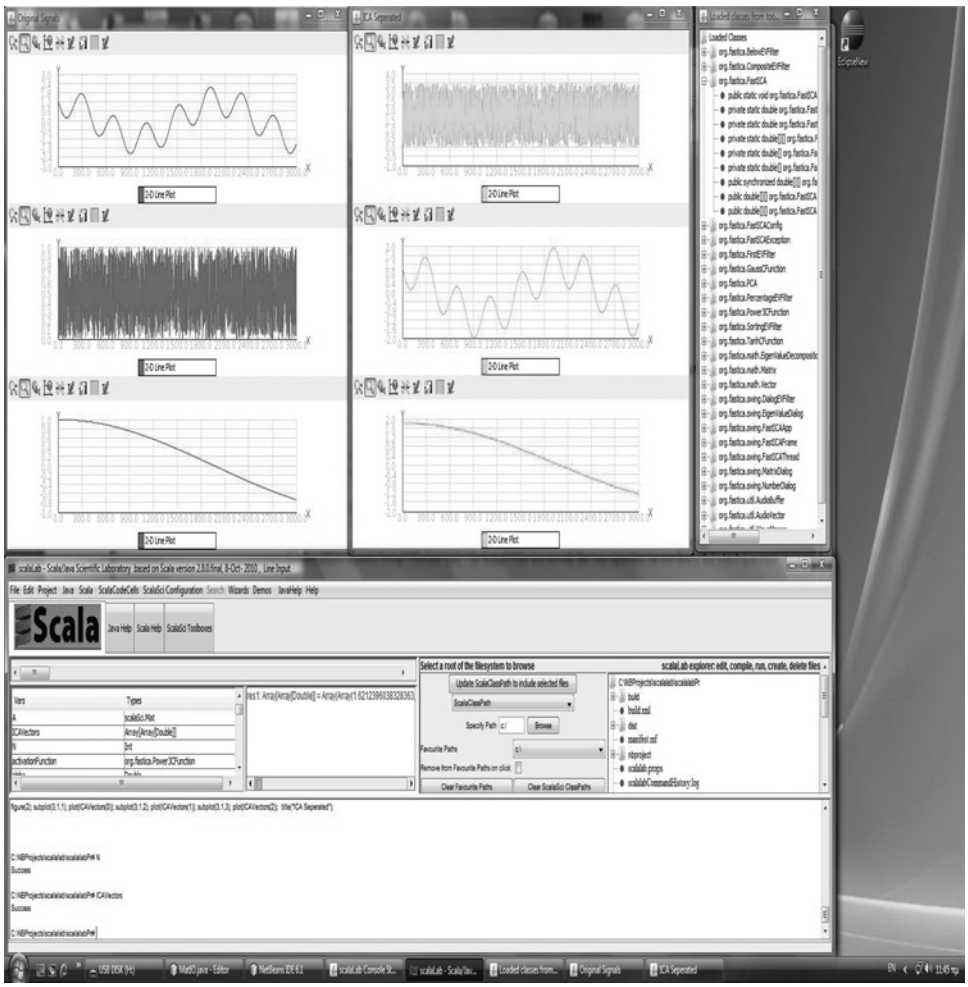

Fig. 4 *Screen snapshot illustrating the application of the Java ICA toolbox to perform signal seperation from within ScalaLab*

A critisism of JVM with respect to number crunching is that is relatively slow. However, for the recent versions of the JIT compiler this no longer holds. Instead, we are impressed that Java outperforms usually C++ in a number of benchmarks we have tested and it is slightly defeated only if C++ is supported with an optimised compiler. Clearly, the optimisations that are performed by the JIT compiler are quite sophisticated, for example, array bounds check elimination [13] and the JVM performance on numeric computations is superb.

For example, we have performed a matrix multiplication benchmark to exploit the relative performance of C++, Java and Scala compilers. We tested gcc 3.4.2 running on SuSE Linux, Visual C++ of Visual Studio 2005, Java 6 and Scala 2.8 with a P-Quad 2.9 GHz with Windows Vista 64-bit. The results for integer and double arithmetic that we have obtained are presented below in table format (Table 1). Thus, the JVM is no longer 'slow' for scientific computation, and further improvements in JIT techniques will make it even better. This fact encourages the software research to improve on scientific programming environments for the JVM.

The jLab environment [1] is similar in style with ScalaLab, but explores the Metaobject protocol of Groovy [3] in order to build high-level mathematical operators. The statically typed Scala language cannot implement a Metaobject-like mechanism, since the later implies the construction of the code for the proper method invocation at run time. Scala therefore builds different mechanisms that provide similar and even better flexibility. We note that with Scala we can implement easily Matlab-like submatrix extraction and access operations with the clever apply and update methods. With Groovy the same requires much more chores to perform.

What is more important is that the resulting numerical code runs with speeds comparable to native Java classcode. In comparison some dynamic languages as, for example, Groovy, although they utilise the effective technique of 'call-site caching' [14], to speed up dynamic calls, are significantly slower. As we tested, with small numerical benchmarks executed both in Groovy based jLab and with ScalaLab, depending on the properties of the code, Groovy is about 40–150 times slower than Scala. It is interesting to note that an open-source project, the Groovy++ project (http://code.google.com/p/groovypptest/), designs a language that permits statically typed code parts mixed with dynamically typed ones. However, although the statically typed code parts are no longer slow, complexity is introduced by this mixed mode execution that at least up-to-now is not concealed from the user.

The 'Scalala' (http://code.google.com/p/scalala/) project created by Daniel Ramage is another similar project with ScalaLab. Scalala exploits also the scalability and flexibility of the Scala language. However, the important difference is that Scalala is mainly a set of Scala mathematical libraries and not an easy to use integrated environment as ScalaLab is.

The Table 2 below compares some attributes of the ScalaLab environment related to other similar environments.

doi: 10.1049/iet-sen.2010.0135

```
// Creates a FastICA standard configuration:
import org.fastica._
var N =3000  // signal size
var t = linspace(0, 2, N)  // linearly spaced samples
var sig = sin(23.4*t)+1.23*cos(4.663*t)  // first signal
var sig3 = cos( 1.2*t)   // second signal
var noise = vrand(N)   // third signal is the noise component
var initialSigs  = new Array[Array[Double]](3, N)
initialSigs(0) = sig
initialSigs(1) = noise
initialSigs(2) = sig3
var numICs = 3
// mixing matrix
var A = new Mat(3,3);
A(0,0)=2.1; A(0,1)=1.1; A(0,2)=0.3;
A(1,0)=0.1; A(1,1)=0.5; A(1,2)=0.4;
A(2,0)=0.12;  A(2,1) = -0.23; A(2,2)=1.2;
// apply source mixing matrix to mix signals
var mixedSignal = A*initialSigs
var sensorMat = mixedSignal.getv  // get matrix as double[][]
var config = new FastICAConfig(numICs,  FastICAConfig.Approach.DEFLATION, 0.01, 1.0e-17,
100000, null)
// build the progress listener
var listener = new ProgressListener()
        {
def progressMade(state: org.fastica.ProgressListener.ComputationState,  component: Int,  iteration: Int,
maxComps: Int) =
                {
println( "\r" +  Integer.toString(component) + " - " + Integer.toString(iteration) + "     ");
                }
        }
// perform the independent component analysis
println("Performing ICA");
var activationFunction = new org.fastica.Power3CFunction()
var filter = new org.fastica.CompositeEVFilter()   // the compositeEVFilter can be used to build a chain
of eigenvalue filters
filter.add(new org.fastica.BelowEVFilter(1.0e-12, false)) // The BelowEVFilter filters all eigenvalues,
which are lower than a given value.
filter.add(new org.fastica.SortingEVFilter(true, true))  // the SortingEVFilter does not really filter some
eigenvalues, but sorts them by their related eigenvectors.
var fica = new  FastICA(mixedSignal.getv, config, activationFunction, filter, listener)
var ICAVectors= fica.getICVectors
figure(1); subplot(3,1,1); plot(sig); subplot(3,1,2); plot(noise); subplot(3,1,3); plot(sig3);
title("Original Signals");
figure(2); subplot(3,1,1); plot(ICAVectors(0)); subplot(3,1,2); plot(ICAVectors(1)); subplot(3,1,3);
plot(ICAVectors(2));  title("ICA Seperated");
```

Fig. 5 *Code of the FastICA example*

At this point we should note that we are in the process of developing a lot of facilities within the ScalaLab framework that will facilitate the development of scientific software. We already have working versions on these user interface components and we improve continuously on them. Specifically, ScalaLab provides:

- an extensive on-line help system based on the JavaHelp (http://javahelp.java.net/) framework;
- keyword sensitive help activated with the F1 function key that is based on the dynamic acquition of the available classes, methods and fields;
- TAB code completion that facilitates the user by presenting the identifiers in-scope and their fields and methods;
- graphical control of the classpath;
- a text editor with code-colouring facilities and the ability to compile Java and Scala sources with source level debugging abilities (e.g. a limited support of breakpoints and 'run-to-cursor');

Table 1 Benchmarking results for performance on numerically intensive tasks

	gcc, ms	VC++, ms	VC++optimised, ms	Java, ms	Scala, ms
integer arithmetic	5554	6021	1981	3807	3634
double arithmetic	5694	6530	1482	3900	3931

We observe that only with code optimisation C++ can outperform JVM code

Table 2 Comparison of ScalaLab attributes related to other similar environments

	ScalaLab	Matlab	SciLab	jLab
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 build-in routines are highly optimised, overall ScalaLab and Matlab run at comparable speeds and which one outperforms depends on the case	much slower than ScalaLab (or Matlab), about 20–100 times slower	slower than ScalaLab, about 5–30 times slower
portability	very portable, anywhere exists installed Java 6 JRE	there exist versions for each main platform, e.g. Windows, Linux, MacOS	there exist versions for each main platform, for example Windows, Linux, MacOS	very portable, anywhere exists installed Java 6 JRE
open-source	yes	no	yes	yes
user-friendness	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 specialised 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 extensiible	SciLab is not designed to be extensiible.	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	jLab is a new project, and thus up-to-now lacks a large user base

• a specialised file system explorer that provides a lot of useful operations on files (e.g. create, delete, edit, compile, run, add to classpath).

Although, such facilities usually exist in modern general purpose integrated development environments (IDEs), for example, in Netbeans, Eclipse, we try to integrate them in ScalaLab more than in any known scientific programming environment.

7 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 specialised Java scientific library can be explored from within Scalalab much more effectively and conveniently.

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.

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 utilisation of these rather complicated libraries.

doi: 10.1049/iet-sen.2010.0135

8 References

1 Papadimitriou, S., Terzidis, K., Mavroudi, S., Likothanasis, S.: 'Scientific scripting for the Java platform with jLab', *IEEE Comput. Sci. Eng. (CISE)*, 2009, **11**, (4), pp. 50–60
2 Papadimitriou, S., Terzidis, K., Mavroudi, S., Likothanasis, S.: 'ScalaLab: an effective scientific programming environment for the Java platform based on the Scala object-functional language', *IEEE Comput. Sci. Eng. (CISE)*, 2011, **13**, (5), pp. 43–55
3 Konig, D., Glover, A., King, P., Laforge, G., Skeet, J.: 'Groovy in action' (Manning Publications, 2007)
4 Odersky, M., Spoon, L., Venners, B.: 'Programming in Scala' (Artima Press, 2008)
5 Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: 'Numerical recipes in C++, the art of scientific computing' (Cambridge University Press, 2002, 2nd edn.)
6 Horstmann, C., Cornell, G.: 'Core Java 2', Vol I Fundamentals, Vol II – Advanced Techniques (Sun Microsystems Press, 2008, 8th edn.)
7 Wampler, D., Payne, A.: 'Programming Scala' (O'Reily, 2009)
8 Subramaniam, V.: 'Programming Scala – tackle multicore complexity on the Java virtual machine' (Pragmatic Bookself, 2009)
9 Lau, H.T.: 'A numerical library in java for scientists and engineers' (Chapman & Hall/CRC, 2003)
10 Haykin, S.: 'Neural networks and learning machines' (Pearson Education, 2009, 3rd edn.)
11 Papadimitriou, S.: 'Scientific programming with Java classes supported with a scripting interpreter', *IET Softw.*, 2007, **1**, (2), pp. 48–56
12 Papadimitriou, S., Terzidis, K.: 'jLab: integrating a scripting interpreter with Java technology for flexible and efficient scientific computation', *Comput. Lang. Syst. Struct.*, 2009, **35**, pp. 217–240
13 Wurthinger, T., Wimmer, C., Mossenblock, H.: 'Array bounds check elimiation for the Java Hotspot Client Compiler'. PPPJ 2007, Lijboa Portugal, 5–7 September 2007
14 Aho, A., Lam, M.S., Sethi, R., Ullman, J.D.: 'Compilers, principles, techniques, & tools' (Addison-Wesley, 2007, 2nd edn.)