Scientific Programming 20 (2012/2013) 379–391
DOI 10.3233/SPR-130359
IOS Press

The software architecture for performing scientific computation with the JLAPACK libraries in ScalaLab

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

[a] *Department of Information Management, Technological Educational Institute of Kavala, Kavala, Greece*
E-mail: sterg@teikav.edu.gr

[b] *Department of Computer Engineering and Informatics, University of Patras, 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, Patras, Greece*

Abstract. Although LAPACK is a powerful library its utilization is difficult. JLAPACK, a Java translation obtained automatically from the Fortran LAPACK sources, retains exactly the same difficult to use interface of LAPACK routines. The MTJ library implements an object oriented Java interface to JLAPACK that hides many complicated details. ScalaLab exploits the flexibility of the Scala language to present an even more friendly and convenient interface to the powerful but complicated JLAPACK library. The article describes the interfacing of the low-level JLAPACK routines within the ScalaLab environment. This is performed rather easily by exploiting well suited features of the Scala language. Also, the paper demonstrates the convenience of using JLAPACK routines for linear algebra operations from within ScalaLab.

Keywords: Java, Scala, functional languages, scripting, interpreters, MATLAB, scientific programming, class loaders, binding

1. Introduction

Fortran 77 was a very popular language for numerical calculations due to its high-performance compilers and intrinsic support for matrices and mathematical functions. However with the increasing complexity of applications and the demand for interactivity it has become necessary to exploit more sophisticated programming environments in order to better manage this complexity.

Object-oriented languages such as C++ and Java were the first major steps to this end. Scripting scientific languages such as MATLAB performed a next major step, towards the interactivity direction. Recently, powerful object-oriented languages like Groovy and Scala offer elaborate scripting facilities and elaborate IDEs. Therefore, the potential to combine both the convenience of scripting and IDE support and the powerful development base of object-orientation naturally arises in the scientific computing domain.

Recently, we introduced the Scala based ScalaLab [14] environment for the Java Virtual Machine. ScalaLab exploits an extended version of the powerful Scala object-functional language [10]. The scientific programming extensions to Scala are referred as the *ScalaSci* language. It presents a MATLAB-like style of working, and compiles the scripts for the JVM. The Scala interpreter implements an elaborate binding scheme that presents not only the data variables but also function and object definitions.

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 and is described in [14]. Also, [15] describes many general aspects of interfacing scientific libraries in ScalaLab. The present paper focuses in detail on the JLAPACK library which has Fortran roots. Therefore it is much more difficult to design clear, easy to use, object oriented interfaces, compared with pure Java numerical libraries. The latter usually already define well organized class hierarchies. Moreover, the current pa-

*Corresponding author: Stergios Papadimitriou, Department of Information Management, Technological Educational Institute of Kavala, 65404 Kavala, Greece. E-mail: sterg@teikav.edu.gr.

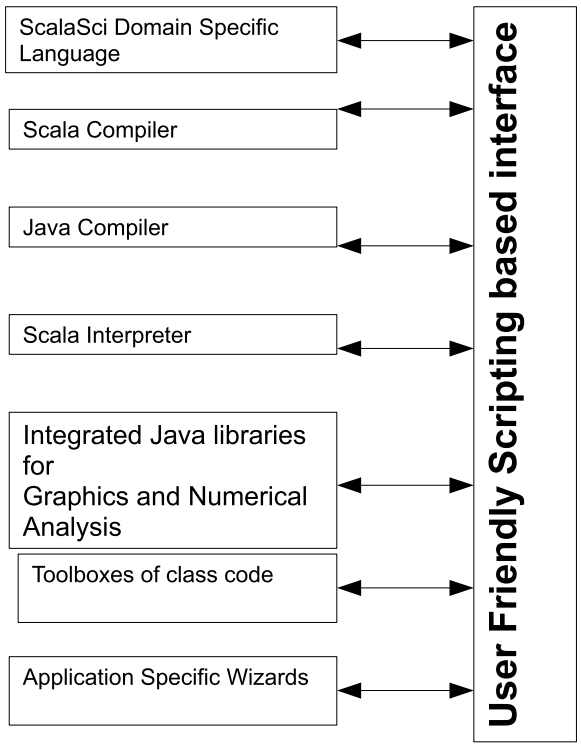

Fig. 1. The architecture of the main software components of ScalaLab.

per focuses on the recent design of the ScalaLab scientific libraries framework, that has some improvements related to the one described in [15], although the general architecture is conserved.

In this article we concentrate on the important subject of interfacing the JLAPACK scientific package in ScalaLab. These libraries are incorporated within the core of ScalaLab. The aim is to provide an easy to use interface to JLAPACK, without compromising its effectiveness. Convenient syntax features, like high-level mathematical operators are implemented by exploiting the rich support that Scala provides. JLAPACK is integrated within the core of ScalaLab. Therefore its sophisticated routines are already available to the ScalaLab user.

The paper proceeds by presenting the general architecture with which ScalaLab exploits effectively scientific libraries for the JVM in Section 2. Section 3 reviews key concepts of LAPACK and BLAS. Section 4 summarizes the MTJ (Matrix Toolkit for Java) library that is a Java high-level front end to some JLAPACK functionality. Section 5 describes aspects of the object oriented interface to JLAPACK that MTJ provides. Section 6 explains the mechanisms of interfacing ScalaLab with MTJ. The implicit conversions feature of Scala is essential and is essential and is described in Section 7. The powerful JLAPACK routines can be exploited directly from ScalaLab without the MTJ agent. Section 8 describes this issue. Section 9 compares aspects of the performance and functionality of ScalaLab with similar systems such as SciLab and MATLAB. Finally, we present the conclusions along with directions for future work.

2. The general architecture of interfacing Java Scientific Libraries with Scala

The general architecture of interfacing Java libraries is illustrated with Fig. 2. The *Wrapper Scala class* (*WSC*) provides a simpler interface to the essential functionality of the Java library, e.g. for matrices A and B, we can add them simply as $A + B$, instead of using the cumbersome Java like method call $A.plus(B)$. For example, some of these wrapper Scala classes are the class *Matrix* for one-indexed matrices based on the NUMAL library [9], class *Mat* a zero-indexed matrix based on Scala implementations that borrow functionality from the JAMA Java package [http://math.nist.gov/javanumerics/jama/] and the MTJ.Mat class based on the MTJ (Matrix Toolkit for Java, http://code.google.com/p/matrix-toolkits-java/) library. Also, these wrapper classes perform the useful task of transforming interfaces to a common pattern, since each Java matrix library has its own style of parameter passing (e.g. returning eigenvalues and eigenvectors). We have to adopt a single one interface (preferably MATLAB-like) in order not to confuse the user.

At this point we should note that although we have a single one-indexed matrix class implementation (coupled with the NUMAL library) we have many possible zero-indexed ones (e.g. for EJML, MTJ, Apache Commons etc.). Overloaded routines such as *sin*(B), can be resolved by the compiler based on the type of B. For example, if B is a one-indexed matrix of type Matrix (based on the NUMAL library), the method *scalaSci.Matrix.sin*(B) is called, if B is an MTJ matrix then the *scalaSci.MTJ.Mat.sin*(B) is called.

By convention important utility routines that return Matrix objects, and end with 1, as *rand1*(n, m), *ones1*(n, m) etc. operate on the one-indexed matrix class. Similarly, those ending with 0, i.e. *rand0*(n, m), *ones0*(n, m) are based on a zero-indexed matrix library. Since we have only a single one-indexed matrix class, the compiler can unambiguously identify the return type, e.g. *rand1*(n, m) returns an $n \times m$ *scalaSci.Matrix* (i.e. one indexed) type. However, this

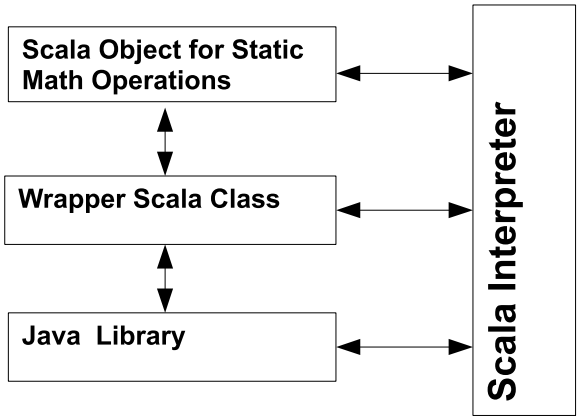

Fig. 2. The general architecture of interfacing Java libraries.

is not the case for the many possible types of zero indexed *Mat* types. For example, an ambiguity arises from the statement:

$$var\ x = rand0(n, m)$$

since all the libraries have a *rand0* method each one returning a different matrix object.

Therefore, it is not correct to have multiple zero-indexed matrices imported at the same time since these utility routines should cope with a particular matrix type, i.e. with the appropriate matrix type representation of the currently utilized library. For example, *rand0*(n, m) returns an MTJ matrix filled with random values when the Scala interpreter is initialized from the MTJ library.

The interpreter needs to switch between different choices for the zero-indexed matrix class. A ScalaSci script that uses only methods of the *scalaSciMatrix* trait (i.e. does not use library specific functions) can be executed by exploiting different libraries, presenting different performances also. Technically, the ScalaSciMatrix trait factors out the common functionality that all the ScalaSci matrix types are enforced to implement.

The *Scala Object for Static Math Operations* (*SOSMO*) aims to provide short overloaded versions of the basic routines for each relevant type. For example, it allows one to use *sin*(B), where B can be an object of our *scalaSci.MTJ.Mat* Scala class, instead of the longer *scalaSci.MTJ.Mat.sin*(B).

Each such 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 interpreter simply needs to import the corresponding SOSMOs in order to switch functionality.

Also, 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 specially designed SOSMOs. For example, there exists an SOSMO object *StaticMathsJAMA* that performs important initializations for the JAMA library and a *StaticMathsMTJ* for the Matrix Toolkits for Java Library one. The utilization of the JAMA library is accomplished by creating a Scala Interpreter that imports the *StaticMathsJAMA* object while for the MTJ the *StaticMathsMTJ* is imported. The ScalaLab user can easily switch different underlying Java libraries. The integrated libraries have strengths in some aspects and weaknesses in others. For example EJML [http://code.google.com/p/efficient-java-matrix-library/] is a fast and elegantly designed library for linear algebra, but it lacks the rich set of functionality provided by the Fortran subroutines styled JLAPACK. Although the particular characteristics and potentialities of each Matrix library differ, for the basic functionality ScalaLab enforces a uniform interface by means of the aforementioned *ScalaSciMatrix* trait (traits in Scala extend the Java interfaces functionality).

The *Java library* module in Fig. 2 corresponds to the Java code of the library. This code performs the main numerical chores. We should note that the Scala interpreter can also use the native Java interface of each library. This is particularly useful for the JLAPACK library. The MTJ library wraps only a small but essential part of LAPACK's functionality. The ScalaLab class *scalaSci.MTJ.Mat* explores more conveniently a part of the MTJ functionality. Therefore, the lower level JLAPACK is much more richer. It can be used from ScalaLab, although with its cumbersome function call interface.

3. LAPACK and BLAS

One of the most significant developments in numerical linear algebra software has been the Basic Linear Algebra Subroutines (BLAS) and the Linear Algebra Package (LAPACK) which builds on BLAS, both written in traditional Fortran 77. The BLAS contain fundamental operations for working with matrices and vectors, such as addition, multiplication, rank-updates and triangular solvers, and this functionality is available for several types of matrices, including general dense, banded, symmetric and others.

Since BLAS has become an important part of many numerical libraries, vendors have created highly tuned versions which can exploit hardware specific details. Such optimized BLAS can achieve theoretical optimal performance, especially for demanding operations such as matrix/matrix multiplications. The BLAS are designed to exploit block algorithms and special matrix types (e.g. tridiagonal, symmetric) [4]. Thus, although with automatically produced Java implementations, these algorithms usually perform significantly better than hand-crafted implementations that do not exploit block algorithms and the properties of special matrices.

LAPACK [2] builds on the BLAS by adding many common matrix solvers and factorizations, including LU and Cholesky solvers, least squares methods and eigenvalue and singular value decompositions. The algorithms used by the LAPACK subroutines are typically the state of the art, and most other matrix software uses portions of it. LAPACKs performance depends on a tuned BLAS; using a standard BLAS can often reduce performance by an order of magnitude.

In order to benefit from the speed and reliability of BLAS, we need a means to call such codes from the Java environment. The Java Native Interface [7] (JNI) provides a portable mechanism to accomplish this, whereby one creates a Java method signature without implementation, then provide an implementation of that method in a separate C source file. As both BLAS and LAPACK are large bodies of code (about 800,000 lines of code in hundreds of interdependent functions), it is impractical to create a replacement in Java, without the assistance of an automated tool.

JLAPACK is a machine translation of the whole set of BLAS and LAPACK into Java using the tool *f2j* (Fortran to Java) [5]. As JLAPACK cannot take advantage of an optimized BLAS (it uses the reference Netlib BLAS), it is typically slower than the native LAPACK.

However, both the advances in JIT compilation and the increase of computing speed make pure Java code suitable for practical numerical number crunching. Additionally, we can obtain some performance advantages by hand crafting some critical Java code, instead of relying only on the automatically translating code of the JLAPACK. Currently in ScalaLab we rely on the pure Java implementation of JLAPACK (i.e. the JNI is not utilized) for portability and simplicity. However, we have an option of using some fast operations using the JBLAS library (http://jblas.org/) that offers conveniently precompiled native BLAS code. Even though JBLAS is easy to use and works transparently, platform dependent problems are always possible in the process of linking calls of Java to native code. For example, while for Win32 platforms all the JBLAS based routines execute perfectly, but for Win64 there are problems. Therefore, we keep JBLAS routines as an option and not in the mainstream.

The Netlib API obtained from the open-source project *netlib-java* (http://code.google.com/p/netlib-java/) provides a Java interface to the JLAPACK functionality. In turn, the Matrix Toolkits for Java (MTJ) project provides a higher level API and is suitable for programmers who do not specifically require a low level Netlib API.

LAPACK on which much of the functionality of MTJ builds upon has three levels of routines [2]

- *driver* routines, each of which solves a complete problem, for example solving a system of linear equations, or computing the eigenvalues of a real symmetric matrix,
- *computational* routines, each of which performs a distinct computational task, for example an LU factorization, or the reduction of a real symmetric matrix to tridiagonal form. Each driver routine calls a sequence of computational routines,
- *auxiliary* routines, which in turn can be classified as follows:
  (a) routines that perform subtasks of block algorithms, in particular, routines that implement unblocked versions of the algorithms,
  (b) routines that perform some commonly required low-level computations, for example scaling a matrix, computing a matrix-norm,
  (c) a few extensions to the BLAS, such as routines for applying complex plane rotations or matrix-vector operations involving complex symmetric matrices.

From the software engineering point of view this organization of LAPACK facilitates the work with a complicated routine-based (i.e. function-based) library. However, as will become evident from the discussion that follows, wrapping an object-oriented layer to the most common numerical tasks, adds significant benefits in terms of usability without compromising notably the speed.

4. 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 forms), for block matrices and for sparse matrices. Most of the functionality of MTJ is built upon the powerful Java JLAPACK package which is a Java translation of the famous LAPACK package [2].

MTJ uses an object-oriented design for its Matrix classes. For example, the *AbstractMatrix* is one of its basic base classes. Some methods of the AbstractMatrix such as *get*(*int, int*), *set*(*int, int, double*), *copy*(), through an *UnsupportedOperationException* and should be overridden by a subclass. Clearly, such operations should perform *according to the specific storage format* of each AbstractMatrix subclass. For the rest of the methods, the library provides simple default implementations using a matrix iterator. Also, all the direct solution methods should be overridden to compute the solution using an algorithm optimized for the particular matrix format. We note that JLAPACK offers algorithms with significant performance benefits when a specific matrix type is used. MTJ can easily use a different library or a customized implementation of a numerical algorithm if the need arises. For example, we can easily use another library instead of JLAPACK, to compute the eigendecomposition.

Operations such 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. The EVD class performs an appropriate call to the powerful and reliable routines of the JLAPACK library. After performing the eigendecomposition the user can conveniently acquire the results from the EVD object by calling the relevant methods, e.g. *getLeftEigenvectors*(), *getRightEigenvectors*(), *getRealEigenvalues*(), *getImaginaryEigenvalues*() *etc*.

ScalaLab implements 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. The eigendecomposition is performed by factorizing the MTJ matrix representation of the data (routine *factorize*). Then the results are prepared with a convenient Scala tuple for output:

```
// compute the eigenvalue decomposition of a 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 */

  var evdObj
    = no.uib.cipr.matrix.EVD.factorize(m.getDM)
  (evdObj.getRealEigenvalues(),
   evdObj.getImaginaryEigenvalues(),
   new Mat(evdObj.getLeftEigenvectors()),
   new Mat(evdObj.getRightEigenvectors()))
}
```

5. MTJ interfacing to LAPACK

The conventional dense storage format of MTJ is based on the matrix construct of the Fortran language. Fortran matrices are laid out column major, that is, columns follow each other sequentially in memory. In contrast, Java's multidimensional arrays are very different: in essence they are arrays of arrays stored row wise. Successive rows of Java arrays are allocated independently and therefore are stored in non-contiguous memory chunks. This is a major obstacle for the optimization algorithms that compilers attempt to apply. The MTJ library uses a one dimensional Java array of doubles to simulate Fortran two dimensional matrices. This array is accessed with column major constructs, i.e., for N rows, the i, j element is stored at the $i + jN$ position of the 1D array, i.e.:

$$A_{ij} = A[i + jN].$$

Also, MTJ explores the properties of special categories of matrices such as packed matrices and banded matrices in order to represent them more effectively. In turn, JLAPACK offers much more effective algorithms concerning numerical tasks on special matrices from the general case. For example, on some tests with symmetric banded matrices of sizes 800×800, the eigendecomposition yielded execution times more than 30 times faster than the eigendecomposition of the same matrix treated as a general dense matrix.

JLAPACK is a powerful package but relatively difficult to use. The JLAPACK translation, although it is performed automatically is relatively fast. We have tested for example, a pure Java implementation of the

BLAS level 3 matrix multiply operation and it is only about 30% faster than the automatically translated one. At this point, we should stress that JLAPACK operates generally well in comparison with pure hand-crafted Java libraries, especially for large matrix sizes. For example, although the JAMA based pure Java eigenvalue decomposition outperformed JLAPACK for matrix sizes smaller than about 50×50, the opposite happens for larger matrix sizes. The rationale of this rather unexpected observation, is of course the quality of the algorithms implemented in the LAPACK package. However, the Java automatically translated code is not at all readable (even worse than the Fortran code!). Also, since Fortran 77 goto statements cannot be implemented always with control constructs in Java, a special class bytecode level tool is required to process the incorrect bytecode in order to implement the goto functionality [5].

The MTJ library implements an object oriented framework around the JLAPACK library. The main classes of this library are illustrated in Fig. 3. Specifically, at the top is the generic *Matrix* interface. It specifies methods which retrieve entries, sets entries, perform matrix addition, multiplication etc.

The abstract class *AbstractMatrix* implements the *Matrix* interface, but does not specify how the matrix is stored. Instead, it provides implementations of most of the methods specified with the Matrix interface by means of elemental access operations and iterators. Therefore, subclasses can only implement those methods which they can perform efficiently, and can delegate the rest to *AbstractMatrix*.

Extending from *AbstractMatrix* are a set of matrices which specify an effective storage layout as well as the corresponding elementary access operations, but not the algorithms themselves. Those provide *get* and *set* methods for working with the entries of the matrix. So whether the matrix is banded, packed or dense,

$$\textit{double Matrix.get}(\textit{int } i, \textit{int } j)$$

always retrieves A_{ij}, thereby hiding any perhaps complicated underlying details. Although, special types of matrices can optimize storage, the interface remains the same with the *get*(), *set*() routines.

The proper LAPACK and BLAS algorithms are chosen by specific matrices such as *DenseMatrix* (a general matrix) and *TriangPackMatrix* (a triangular matrix stored in packed format). For instance, direct solvers are different for these formats, the former must use a general LU with partial pivoting while the latter can be solved without factorizing. A remarkable achievement of the MTJ library, is that it masks such differences away, and one can solve $Ax = b$ by calling

$$\textit{Vector Matrix.solve}(\textit{Vector } b, \textit{Vector } x);$$

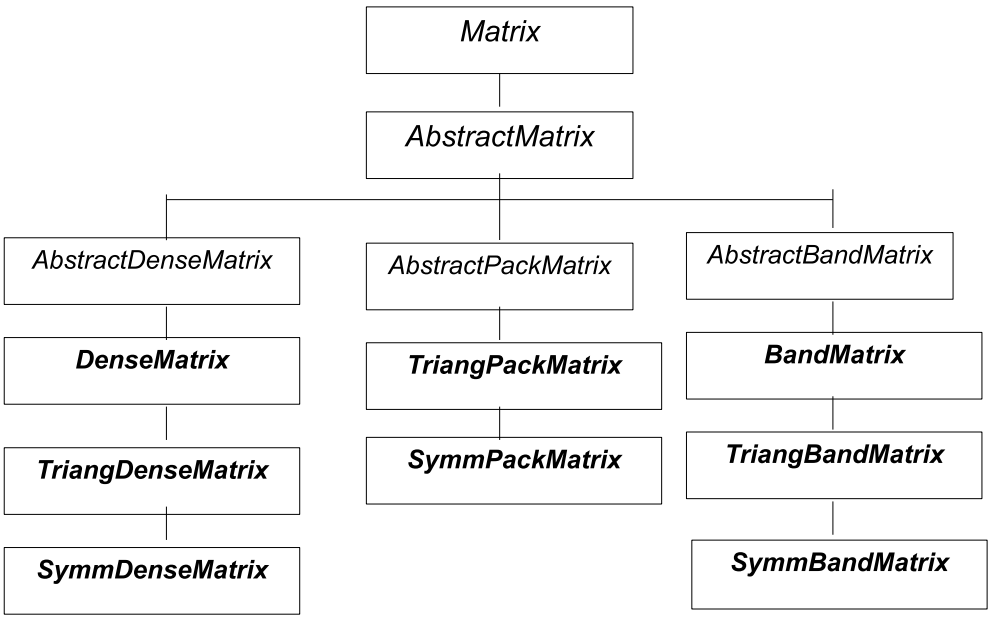

Fig. 3. The main parts of the matrix hierarchy of the MTJ library.

The corresponding LAPACK calls are much more involved, and are different for each matrix type.

A significant characteristic of the MTJ library is the *matrix iterators*. A matrix iterator passes over all the stored matrix entries, returning one entry at a time. The details of this traversal are matrix specific. Iterators are of fundamental importance in processing sparse matrices. They exploit sparsity naturally, since they traverse only the relevant parts of the matrix.

6. Interfacing the MTJ library in ScalaLab

This section describes some features of the Scala language that facilitate significantly the utilization of Java Scientific Libraries. It concentrates specifically on the JLAPACK library.

Below we elaborate on the approach of wrapping the MTJ library with a Scala class. This class is the MTJ.*Mat* class.

In Scala operations on objects are implemented as *method calls*, even for primitive objects like Integers. However 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. *var* $b = a * 5$ corresponds to *var* $b = a.*(5)$. Scala makes it 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 that performs Matrix transposition. Doing that we can write the transpose of A as $A\sim$.

We implement in Scala syntactically elegant indexing on any objects with the *apply* method and assignment of values with the *update* method. For the *Mat* class, obviously we want if M is a Mat to access its (i, j)th element as $M(i, j)$. Thus we implement the apply method with the following code pattern:

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

We note that the *apply* method calls the corresponding routine *get* of the MTJ library. The Scala compiler supports flexible syntax for the *apply* and *update* methods, e.g. we can call $M(i, j)$ instead of $M.apply(i, j)$ and write: $M(i, j) = 9.8$ 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 = {
  dm.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, e.g. $M(2, k, m, 4, 2, N)$ to extract a range denoted in MATLAB as $M(2{:}k{:}m, 4{:}2{:}N)$.

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 in Java Virtual Machine programming can be implemented easily with *static imports*. In Scala, we import objects since these encapsulate the 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 into the global environment only the object and not the particular method.

7. Implicit conversions: An important feature of Scala for convenient syntax

Returning to our matrix *Mat* class example, 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 situ-

ation when a Mat is added to a double array. Dynamic languages as Groovy [8], 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 [10,17,18] 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*, *RichDouble2DArray* and *RichDouble2DArray* classes.

Initially, we have implemented implicit conversions that transformed the receiver object according to the type of the arguments in order for the operation to proceed, e.g. for the code below:

var $a = rand(200, 300)$ *// create a 200 by 300 Matrix*
var $a2 = 2 + a$ *// performs the addition by implicitly converting 2*

The 2 at the initial design was transformed to a 200×300 Matrix filled with 2, and the addition operation is performed as Matrix addition. However, this design was not very elegant and also not as efficient in terms of speed and memory usage as it can be.

Therefore, we updated the design of the implicit conversions 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 2 is transformed by the Scala compiler to a *RichNumber* object, that defines an operation to add a Matrix. Therefore, the operation proceeds effectively without allocating any new space (at the initial design a Matrix object was created and filled with 2 s).

This design is more effective (about 20–30% speed increase) and (perhaps more important) simpler and more extendable.

We present a small part of the *RichNumber* class for illustration (Listing 1).

Listing 1. Part of the RichNumber class to which matrix objects from library classes are implicitly converted

```
// e.g. 2 + Mat
package scalaSci
class RichNumber(v: Double) {
private val value = v // the RichNumber includes a
Double field that corresponds to its value

// follow many methods here
//...

// addition of RichNumber and Vec
def + ( that: Vec ): Vec = {
var N = that.length
var result = new Vec(N)
var r = 0
while (r < N) {
result(r) = that(r) + value
r+ = 1
}
result
}

// also follow many methods here
// ...
}
```

Similarly, the classes *RichDouble1DArray* and *RichDouble2DArray* wrap the *Array*[*Double*] and *Array*[*Array*[*Double*]] Scala classes in order to allow convenient operations as 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.

The implementation of the *RichDouble1DArray*. *scala* and *RichDouble2DArray.scala* classes can be obtained from the sources of ScalaLab. We next describe some aspects of extending the functionality of these basic classes using JLAPACK.

8. Extending the functionality of the *RichDouble2DArray* object with JLAPACK

Although MTJ presents much of the functionality of JLAPACK in a much easier to use object-oriented way, the user has to perform some MTJ specific chores (e.g. to create and initialize properly an MTJ DenseMatrix) and to study the architecture design of the MTJ and its API before using it effectively. Also, much of the JLAPACK's potentiality is not wrapped by MTJ.

These facts motivated the creation of an alternative interface to JLAPACK that acts directly on the standard *Array*[*Array*[*Double*]] type and injects a lot of linear algebra manipulation routines into this standard Java/Scala type. Note also that the implicit conversion of *Array*[*Array*[*Double*]] to *RichDouble2DArray* allows the MATLAB-like matrix handling of Java's 2D double arrays within ScalaLab.

For example, consider the implementation of the eig(a) routine that provides the real and imaginary eigenvalues and the corresponding right-eigenvectors of an Array[Array[Double]]. We can observe that although the routine is complicated and its understanding requires knowledge of the LAPACK internals, the interface of the routine to the ScalaLab user is very simple and MATLAB like. The routine simply accepts a Java square 2D array of doubles (i.e. the matrix), and returns a tuple consisting of: (a) the real parts of the eigenvalues of the matrix, (b) the corresponding imaginary parts and (c) the eigenvectors of the matrix.

```
// computes eigenvalues and right eigenvectors
def eig(inM: Array[Array[Double]]) = {
val n = inM.length
// Allocate space for the decomposition
var Wr = new DenseVector(n)
var Wi = new DenseVector(n)
var Vr = new DenseMatrix(n, n)
// Find the needed workspace
val worksize = Array.ofDim[Double](1);
val info = new intW(0)
LAPACK.getInstance.dgeev(
   "N", // left eigenvectors of A are not computed
   "V", // right eigenvectors of A are computed
   n, // the order of the matrix, number of rows
   Array.empty[Double],
   Math.max(1, n), // leading dimension of the array
   Array.empty[Double], // Wr: real parts of the computed eigenvalues
   Array.empty[Double], // Wi: imaginary parts of the computed eigenvalues
   Array.empty[Double], // if JOBVL = 'V' the left eigenvectors u(j) are stored one after
// another in the columns of VL, in the same order as their eigenvalues
// if JOBVL = 'N'm VL is not referenced'
   Math.max(1, n), // the leading dimension of the array VL
   Array.empty[Double],
   Math.max(1, n),
   worksize,
   -1, // a workspace query is assumed
   info)
// Allocate the workspace
val lwork: Int =
   if (info.'val'! = 0)
   Math.max(1, 4 * n);
else
   Math.max(1, worksize(0).toInt);
val work = Array.ofDim[Double](lwork);
```

```
// Factor it!
val A = new DenseMatrix(inM)
LAPACK.getInstance.dgeev(
  "N", // left eigenvectors of A are not computed
  "V", // right eigenvectors of A are computed
  n, // the order of the matrix, number of rows
  A.getData, // (input/output) array, on entry the n × n matrix A, on exit A is overwritten
  Math.max(1, n), // leading dimension of the array
  Wr.getData, // Wr: real parts of the computed eigenvalues
  Wi.getData, // Wi: imaginary parts of the computed eigenvalues
  Array.empty[Double],
  Math.max(1, n),
  Vr.getData,
  Math.max(1, n),
  work,
  work.length, info);

if(info.'val' > 0)
  throw new
    NotConvergedException(NotConvergedException.Reason.Iterations)
else if (info.'val' < 0)
  throw new IllegalArgumentException()

// prepare the results
var rWr = new RichDouble2DArray(scalaSci.JILapack.denseVectorToDoubleArray(Wr))
var rWi = new RichDouble2DArray(scalaSci.JILapack.denseVectorToDoubleArray(Wi))
var rVr = new RichDouble2DArray(scalaSci.JILapack.denseMatrixToDoubleArray(Vr))
  (rWr, rWi, rVr)

  }
```

In a similar spirit we extend ScalaLab with similar powerful routines based on JLAPACK that are easy to use but rather difficult to implement.

9. Accuracy and performance of the ScalaLab libraries

Java conforms fully to IEEE standard [2] 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. 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 [3]. 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 Native BLAS 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 a Pentium Dual Core machine clocked at 1.5 GHz, running Windows Vista. 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 2–5 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.

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.

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 [16] 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.

Also, another interesting point is the storage format used by each library for matrix access. Some libraries use row-major format (e.g. EJML), others column major (e.g. JLAPACK) and others a 2D Java array. Generally, the column major format seems to be more efficient, but the difference is not significant. The following table (Table 2) presents the basic Java libraries that ScalaLab exploits and their main properties.

10. Conclusions and future work

This paper has presented some ways by which we can work more effectively with the MTJ Java scientific library and the JLAPACK numerical analysis system. We demonstrated that ScalaLab can integrate elegantly these 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.

JLAPACK although it is obtained by automatic translation with *f2j* presents performance that competes with hand crafted Java libraries and for difficult numerical tasks, usually outperforms them. Com-

Table 1
Results of some basic benchmarks

	ScalaLab (s)	SciLab 5.21 (s)	MATLAB 7.1 (s)	GroovyLab (s)
Matrix multiplication	1500 × 1500: 6.64 with Java, 3.06 with Native BLAS 2500 × 2500: 30.02 with Java, 13.74 with Native BLAS	1500 × 1500: 10.9 2500 × 2500: 40.48	1500 × 1500: 2.96 2500 × 2500: 12.8	The same as ScalaLab
2500		30.02	13.74	13.55
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

bined with the state of the art algorithms used in LAPACK and the familiarity of a significant part of the scientific community with LAPACK from Fortran, it makes JLAPACK a strong computation vehicle within ScalaLab. Moreover, by exploiting the JBLAS library, some basic LAPACK operations can be used by the ScalaLab user in their native optimized implementations, in which they are about 2 to 5 times faster, without making anything different from pure Java/Scala programming.

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.

The ScalaLab environment combines the solid and extensible basis of object orientation, the ease-of-use of scripting, the rich Swing graphical environment and the expressiveness of the Scala language. The object-oriented framework built on top of the JLAPACK library, that the article described, is only one possibility

Table 2
The basic libraries used in ScalaLab and their main properties

	Linear algebra only	Extensive coverage of algorithms	Support for special matrices	Efficiently	Storage format
JAMA	Yes	Only the basic operations	No	Implements the basic algorithms, is the slowest library tested	Java 2D array
JLAPACK	Yes	Very extensive coverage	Many special matrices block, triangular, banded	Although automatically produced, is very efficient, especially for large matrices and special matrix types	Java 1D array in column major order
EJML	Yes	Relative many operations	Support for block matrices	For the common problems, is generally the fastest library tested, usually a little faster than JLAPACK	Java 1A array in either row or column major order
Apache	No, covers many numerical analysis tasks	The basic operations	Block algorithms supported	Is a fast library but not for all problems	Java 2D array
NUMAL	No	Very extensive coverage	Banded, triangular, symmetric	Generally, is an efficient library	Java 2D array

of strengthening ScalaLab as a scientific programming environment. A lot of work remains in extending and improving ScalaSci using the philosophy of development that the article has presented.

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. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney and D. Sorensen, *LAPACK Users' Guide*, 3rd edn, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999, available at: http://www.netlib.org/lapack/lug/.

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

[4] J.J. Dongarra, J. Du Croz, S. Hamarling and I. Duff, A set of level 3 basic linear algebra subprograms, *ACM Transactions on Mathematical Software* **16**(1) (1990), 1–17.

[5] D.M. Doolin, J. Dongarra and K. Seymour, JLAPACK-Compiling LAPACK FORTRAN to Java.

[6] B.-O. Heimsund, High performance numerical libraries in Java.

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

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

[15] S. Papadimitriou, K. Terzidis, S. Mavroudi and S. Likothanassis, Exploiting Java scientific libraries with the Scala language within the ScalaLab environment, *IET Software* **5**(6) (2011), 543–551.

[16] 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.

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

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

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