Scientific Computing with ScalaLab at the Java Platform

Table of Contents

What's new

The purpose of this section is to describe in somewhat draft form some important new features, along with the timestamp of their implementation.

March 12

a. Change in the way to multiply matrices

The operator * performs serial multiplication of matrices.

For large matrices the operator *& should be used that performs significantly faster since it exploits both native BLAS and Java multithreading. However, for small matrices the simpler pure Java multiplication performs faster.

Also for addition and subtraction the +& and -& operators exploit Java multithreading.

b. MouseMotionListener is not enabled by default.

Since the plenty of output producing by scrolling the mouse pointer over variables can annoy the user, by default this feature is disabled. However, it can be enabled with the relevant option from the **Configuration** menu.

Preface

The ScalaLab environment builds upon Scala, a modern powerful object-functional language for the Java Virtual Machine with full Java interoperability. The book describes how ScalaLab extends Scala with MATLAB-like constructs in order to implement a powerful compiled mathematical scripting framework. ScalaLab is an integrated scientific programming environment in the spirit of MATLAB. It is not a Scala scientific software library, as for example is Breeze (https://github.com/scalanlp/breeze). In fact, ScalaLab

utilizes many superb numerical software libraries, for example the EJML library (http://ejml.org/wiki/index.php?title=Main_Page), the Apache Common Maths (http://commons.apache.org/proper/commons-math/), the JBLAS library (http://jblas.org/) etc. For many of them ScalaLab builds higher level syntactic sugar in order to allow easier utilization with MATLAB like concise expressions.

ScalaLab aims mainly to utilize scientific software developed for the Java Virtual Machine (JVM). Most of that code is developed with Java, but also compiled classes with other JVM languages (as for example Scala itself) can readily used. Also, experimental optional ScalaLab subsystems explore native C/C++ code, for example the GNU Scientific Library (GSL) (https://www.gnu.org/software/gsl/) and support for GPU (Graphical Processing Unit) NVIDIA CUDA computing (https://developer.nvidia.com/cuda-zone). Although these options can obtain significantly better performance than Java in some specific applications, they lack the portability, simplicity and productivity of the Java platform. We should note that the elaborate design and implementation of the Hotspot JVM performs sophisticated JIT (Just In Time) compilation and rivals the performance of optimized C/C++ code. Therefore, in most practical cases the performance of the JVM scientific libraries is adequate.

The reader should have programming experience preferably with Java or even better with Scala. However, for ScalaLab programming **advanced programming techniques are not required, although they can be used**. One of the main objectives of ScalaLab is to be very **user friendly** and to present a **simple high-level scripting language** to the scientist in the spirit of MATLAB. However, an advanced programmer can also utilize the sophisticated Scala language on which ScalaLab is build upon.

ScalaLab is efficient and can be an interesting open-source alternative to commercial packages, especially for the scientific community familiar with Java. The user interface of ScalaLab explores the potential of Java's Swing library in order to facilitate the work of the scientist. In addition, the user of ScalaLab can mix easily and Java code that can be compiled with the system's Java compiler (i.e. javac). The **javac** compiler whenever is available (i.e. the JDK has installed) can also be used to compile Java code, and can be called directly from ScalaLab. However, unless we have the not too common task of compiling Java sources, installation of the Java JDK is not required to work with ScalaLab (of course, the installation of Java's JRE is necessary).

An extension of Scala with MATLAB-like constructs, called **ScalaSci** is the language of ScalaLab. **ScalaSci** offers MATLAB-like high-level mathematical operators by exploiting the flexibility of the Scala language for building easily new syntax. ScalaSci is effective both for writing small scripts and for developing large production level applications. The book describes the core scientific classes of ScalaSci, and presents examples of their application.

Since Scala is statically typed, ScalaSci code runs at the full Java's speed. Also, since the optimizations performed by the Just-In-Time Java compiler result in code that runs with speed similar to native code, ScalaLab can run heavy numerical tasks with speed competitive to the traditional C/C++ or Fortran compiled code.

The ScalaLab editor is based on the **jsyntaxPane** editor (http://code.google.com/p/jsyntaxpane/) which provides a convenient environment for editing code. In addition, some facilities like code completion, execution of selected text and step to cursor are implemented, so the environment has some of the facilities of an IDE (e.g. Netbeans) integrated with the power of scripting in developing code. An alternative editor with similar capabilities is based on **RsyntaxTextArea (http://fifesoft.com/rsyntaxtextarea/)** component. This editor can perhaps operate better in some platforms and can be preferred by some users. Therefore, there is an option to set it as the preferred editor.

The ScalaLab editor is in continuing development, and currently supports a set of useful functions, such as automatic importing the necessary packages for categories of applications, compile of Java sources with the system's Java compiler (when *javac* is installed), compile of Scala sources with the Scala compiler, single-step ScalaSci code and text coloring. A second supported editor, is the well-kown **jedit** open-source Java programmer's editor, that offers sophisticated editing of Java code. The jedit was integrated in ScalaLab and includes its **Scala plugin.** Therefore, for files with a .scala extension, the integrated jedit of ScalaLab offers some programmer's help, such as text coloring and limited code completion support.

We note also that we have setups for developing standalone Java and Scala applications that use ScalaSci libraries. In these cases the user can work with advanced developing environments as **Netbeans** and **Eclipse** in order to develop Java and Scala applications that use ScalaSci libraries and can be executed standalone without the Graphical User Interface

of ScalaLab.

The **ScalaLab explorer** is a specialized file manager that is built upon Swing and has convenient file handling functionality, such as file browsing, deleting, creating, editing files, along with the potential to compile and execute both Scala and Java code. Also, classpath related operations can be managed from the ScalaLab explorer. Specifically, we can add paths to the Scala classpath, which are available when we create a new Scala interpreter.

By exploiting the flexibility and extensibility of the Scala language we also present the framework for the utilization of Java scientific libraries within the ScalaLab environment. We describe the interfacing of basic Java numerical analysis libraries, as the **NUMAL** library, **MTJ** (Matrix Toolkit for Java) which has an object-oriented wrapping to some **JLAPACK** libraries and significant functionality for **sparse** matrices and **distributed** matrices, the **EJML** (Efficient Java Matrix Library) and the Apache Common Maths library. Although we believe that Scala is a much improved language compared to Java (i.e. something like “Java.next”!) and we impressed by the elegance and sophistication of its design, Java remains a good and effective solution for simple tasks as the development of scientific software. Thus, the philosophy of ScalaLab is not to replace Java, but to exploit also the Java language, both by utilizing the excellent Java scientific libraries and by offering the possibility to the Java programmer to use Java to develop applications, in case where is more familiar with Java. For this reason ScalaLab integrates a Java REPL (Read-Evaluate Print Loop).

In ScalaLab, Java libraries for specific applications can be easily utilized as **toolboxes**. We claim that Java scientific software can be exploited much more easily and effectively from within ScalaLab. Toolboxes are especially useful, since they allow to extend the functionality of ScalaLab to specific application types. Therefore, ScalaLab offers support for installing/removing toolboxes easily with a graphical interface. The installed toolboxes remain at the subsequent sessions. Also, toolboxes placed at the **defaultToolboxes** folder are directly available.

Chapter 1 The ScalaLab environment

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 like MATLAB [6], Maple and Mathematica [3, 4] and open source packages like Scilab [1] and Octave [5].

The book presents an open source mathematical programming environment for the Java Virtual Machine (JVM), the ScalaLab environment [19]. Although ScalaLab presents a MATLAB-like style of working, it differs from the forementioned in that it compiles the scripts for the JVM. ScalaLab utilizes and expands the powerful Scala object-functional language [15].

A critisism of JVM with respect to number crunching is that is relatively slow. However for the recent versions of the Just-In-Time (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 optimized compiler. Clearly, the optimizations that are performed by the JIT compiler are quite sophisticated, e.g. array bounds check elimination [18] and the JVM performance on numeric computations is superb.

Interactive scientific programming environments have gained much popularity mainly for their simplicity and the great speed improvements with JIT 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++ and Java. The later language although is not designed for numerical computation has become very popular due to its portability, reliability, simplicity and the advances in Just In-Time (JIT) Compilation. The relatively new Scala language [15] retains all the benefits of Java while in addition it builds a novel scalable architecture. ScalaLab exploits both the flexibility of Scala and the robust and feature rich Java platform in order to offer an open MATLAB-like system to the scientist. The user of ScalaLab can work both with the MATLAB-like script oriented way or with the Java/Scala like way of building stand-alone applications. Also, the ScalaLab user can replace scripts with classes in order to build gradually a complex application.

In this book we concentrate on two main directions. The first one, describes 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 like high-level mathematical operators are implemented by exploiting the rich support that Scala provides. Many features of Java's Swing 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 utilized library. Specifically, the powerful **RichDouble2DArray** class is a simple wrapper to the **Array[Array[Double]]** type and exploits algorithms from various libraries in order to present an effective and consistent set of routines to the ScalaLab user. These routines seek for a MATLAB-like syntax in order to facilitate the user. However, for some major Java scientific libraries, e.g. NUMAL, MTJ, EJML, there exist Scala wrapper classes that present some of their potential in a higher level and easier to use form. Although, these Scala classes have differences in the details, that are due to the differences of the Java numerical libraries that they wrap, they maintain uniform the basic set of routines, as defined by the **StaticScalaSciGlobal** trait. For example, we can use *rand0(5,10)* independently of the library to create a random matrix of size 5X10 and we can use the \ operator to solve linear systems.

Therefore, ScalaLab abstracts a set of basic routines from all the libraries in a uniform way. A question that can asked at this point is why to keep many libraries. We decided to keep them since each of these good Java libraries has its strengths, for example JLAPACK is powerful for linear algebra, EJML is generally quite fast, Apache Commons and NUMAL support many numerical analysis operations, e.g. they provide ODE solvers, statistical analysis, optimization routines and even machine learning. It is important that each can be easily used from ScalaLab with its Java interface. The later permits to exploit the full potential of the library.

The presentation proceeds by describing first the high-level operators and methods that are implemented in Scala for the wrapper classes of each mathematical library. These operators allow MATLAB-like handling (e.g. addition, subtraction, multiplication etc.) of matrices. For example we can add two matrices A and B, with A+B, perform the operation A*B-4.7*(A+B) etc. Also, convenient Matrix access and update operators are implemented, for selecting submatrices, for example A(5::2::18, 49::-2::5). Moreover, many useful mathematical functions, available at the matrix level, facilitate the user to write high level expressions, e.g. sin(A), cos(A-3.4*B). These operators are common i.e. library independent.

Next, we describe some of the basic each basic libraries independently, presenting examples of their applications.

The second direction is the utilization of external Java libraries of scientific code for toolboxes. The system assists the user by providing insight about the contents of the toolboxes using the interrogation potential of the Java reflection API. We describe examples of utilizing some

toolboxes.

The architecture of the ScalaLab environment

ScalaLab builds upon the technology stack developed over many years for the Java Virtual Machine. ScalaLab can run any Java code compiled with the *javac* compiler, and can also compile Java classes. Evidently, it can compile and run Scala code. Scala classes are advantageous in scripting and faster development. However, coding directly at the Java level, i.e. without using higher-level complicated constructs, can present advantages when we want to produce stand alone applications for the Java Virtual Machine. In that case, using pure Java classes, the application can run without the rather complicated Scala run-time support (something very useful for embedded Java applications).

The main components of that technology are (Fig. 1):

a. The Just-In-Time (JIT) Java Compiler

Although transparent to the user this component is perhaps the most important. Advances in Just-In-Time compilation, allow bytecodes today to execute with speeds that exceed even that of natively compiled code. This fact is critical in the area of numerical code that involves a lot of number crunching. We note that we provide scripts that execute ScalaLab with either a *client* or a *server* **JVM** and sometimes the server JVM has significant improvement in performance for heavy computational tasks.

b. The Scala Compiler and Run Time Libraries

ScalaLab integrates a recent version of the whole Scala system. Thus, a seperate installation of Scala is unecessary.

c. The Scala interpreter

This component is the core of ScalaLab and the one that implements the MATLAB-like sense of the system. It is an advanced interpreter that explores well the internals of the Java Virtual Machine in order to maintain state between successive script executions.

d. The integrated Java Libraries

Java Libraries for plotting and for the most important numerical tasks have been integrated with ScalaLab and are directly available to the user.

e. External class libraries

Since ScalaLab runs upon the Java Virtual Machine any class file can be executed as long as it is placed on the ScalaLab classpath. ScalaLab keeps an adaptable environment variable *ScalaLabClassPath* for implementing the notion of classpath. That variable controls the classpath of the Scala Interpreter.

Special support is offered for conveniently placing .jar files at the ScalaLabClassPath that pack Java libraries, since it is especially important to utilize effectively toolboxes of Java code.

f. Application Level Wizards

These tools greatly facilitate the development of specialized applications. We will present an example of solving Ordinary Differential Equations (ODEs) with ScalaLab, where the wizards are particularly useful.

g. The Computer Algebra Subsystem

Computer Algebra facilities are developed based on the symja (http://code.google.com/p/symja/) open source Java Algebra system.

h. The Imports Wizard

ScalaLab uses a large number of libraries. This wizard allows to perform easily the imports required for each library by injecting the proper import commands. For example, we can easily inject the basic Computer Algebra imports if our application performs Computer Algebra tasks. Also, very useful is the search in jar library files performed by options in the “Utilities” menu. For example, if we search for classes related to “eig” we can write “eig” or “eig.” and use the option of “Search all classpath for injecting the proper import statements”. The proper import statements for classes related to “eig” are automatically injected at the editor’s text. Evidently there can be import statements that are not useful, but all the imports compile, since they refer to classes on the classpath.

The user interface of the ScalaLab environment consists of the following components:

a. The *ScalaLab Interpreter Pane* which is based on the *jsyntaxPane (http://code.google.com/p/jsyntaxpane/)* that offers an IDE like editor, with facilities such as code completion, execution of selected text, run – to – cursor, etc. combined with the power of scripting. A similar interface built around the ***RsyntaxTextArea*** ***(http://fifesoft.com/rsyntaxtextarea/)*** component and can be preferred by some users. These interfaces are the preferred interfaces for writing scripts. However, an additional console based interface exists, the *ScalaLab console.*

b. The *ScalaLab console* receives commands and scripts from the user. Control commands (e.g. ***dir, pwd, cd***) are executed directly, while scripts are issued to the Scala interpreter for execution. A popup menu also displays useful options, such as resetting the Scala interpreter or switching between interpreters that use different Java scientific libraries. The user can copy and paste code at

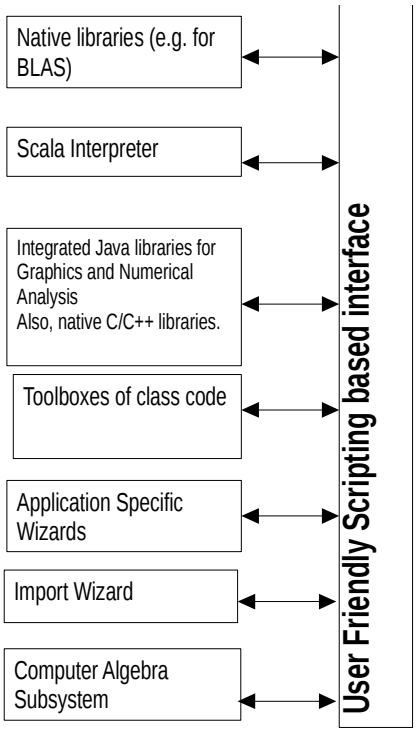


Figure 1 The Architecture of the ScalaLab

the Console window and this code is executed.

It is important to note that both the *ScalaLab Interpreter Pane* and the *ScalaLab Console* share the same Scala interpreter. Therefore, the popup menu of the *ScalaLab Console* can be used to control effectively which interpreter is used, s well as the “Scala Interpreter” top-level menu.

c. The ScalaLab's explorer which allows to browse conveniently the filesystem and to perform operations on files e.g. to edit, compile, run Java and Scala files.

d. The ScalaLab editor, is a simple text editor based on the **jsyntaxpane** component or the ***RsyntaxTextArea***, that however assists by offering special operations such as compiling and running Java/Scala files and with convenient execution facilities for execution of script code (e.g. execute selection, run-to-cursor).

e. The *Console output* window that scrolls the results of the commands. In essence, the Java's **System.out** stream is redirected at the output console and thus all the output can be conveniently examined. Also, the console output window displays all the output of the Scala Compiler that produces when the Scala Interpreter feeds to it the script for compilation.

f. The *variable workspace* window that displays the globally accessible variables and their types. Watching of variables is controlled with an option from the Configuration menu.

ScalaLab is based on .jar libraries of three types:

a. ***The Scala language jars:*** *“**scala-compiler.jar”, “scala-library.jar”, “scala-swing.jar”.*** These are the official Scala Language files implementing the Scala Compiler, Scala Libraries and Scala Swing respectively. In the implementation of ScalaLab we have retained **unaltered** the Scala language implementation. Thus, future versions of Scala can be directly utilized in ScalaLab. ScalaLab can be readily updated to newer Scala versions, simply by replacing these files.

b. ***The ScalaLab auxiliary core numerical libraries files, there are several, and have the core functionality of each library, e.g. EJML, NUMAL, MTJ etc.*** In order to retain the ScalaLab core

small we have placed a few numerical libraries within the core of ScalaLab, and the basic plotting libraries.

c. Auxiliary jars, e.g. Help files, the jEdit jar files etc.

All these libraries are placed within the **lib** folder. However, there are additional .jar files in the DefaultToolboxes folder. These are useful toolboxes, that however can be removed without affecting the operation of ScalaLab.

Comparison with MATLAB

The following table compares some basic features of ScalaLab with those of MATLAB.

Feature	ScalaLab	MATLAB
Speed	Is faster for general scripts. Fast native BLAS operations supported with the JBLAS library obtain about the same speed as MATLAB's built-in operations.	MATLAB wins when fast built-in operations are used.
Rich Extensible Language	Clearly, Scala is a very rich and extensible language.	Limited extensibility using .m scripts.
Plotting	ScalaLab has extensive plotting support using Java libraries. Although all the libraries are not integrated elegantly up-to-now, the basic plotting interface operates similar to MATLAB's plots.	Standardized and well-documented plotting support.
Libraries	ScalaLab is compatible with all the Java scientific libraries.	MATLAB has an extensive toolbox set.
Easy of use	ScalaLab is very user-friendly but still much work needs to be done to "smooth" the user experience.	Capable and productive user interface.
Stand Alone Applications	It is very easy with ScalaLab to produce stand alone versions of the scripts that are executed on the Java Virtual Machine	MATLAB needs separate compilers.

The Scala Interpreter of ScalaLab and the default imports

The Scala interpreter is a sophisticated piece of code, that creates a scripting environment on top of the Scala compiler.

ScalaLab performs a large number of imports in order to create a pre-initialized convenient environment. Specifically the ScalaLab interpreter imports all the necessary imports in order to allow the user to work conveniently (e.g. they can execute directly the ScalaLab demos). These imports include:

a. The general ScalaSci classes: the enhancement of the Array[Array[Double]] with implicit conversions, **Vec, Matrix** (one-indexed matrix class based mostly on the NUMAL library), and the **Mat** (zero-indexed matrix class) appropriate for the utilized library (e.g. JAMA, EJML, MTJ. Apache Common Maths).

b. The wrapper classes for the two main plotting libraries: **jMathPlot** and **jFreeChart.**

c. The I/O classes (e.g. writing to text data files, XML files, MATLAB format support)

d. The NUMAL library imports

e. The imports for the Computer Algebra (sysja) subsystem

Note that when we install toolboxes, the Scala Interpreter should be recreated since we cannot alter the classpath of a running Scala interpreter. In this case the user has the option of creating either a JAMA or an EJML based Scala Interpreter. Afterwards, the user can change the library preferences by creating an interpreter of different type (e.g. of MTJ type). The installed toolboxes remain on the classpath independently of the created interpreter type. Also they are available on subsequent ScalaLab sessions, unless they are explicitly uninstalled. However, an easier way to install toolboxes is simply to drop them within the *DefaultToolboxes* folder.

The importHelper object

The importHelper object facilitates the user to import the required staff for particular libraries. It implements methods that perform the required import statements using the current interpreter.

Therefore, after the execution of these statements the corresponding imports are available. For example, in order to conveniently use the Java Swing routines, the *importJavaSwing* method interprets the corresponding imports:

```
 def importJavaSwing() = {
    GlobalValues.globalInterpreter.interpret("""
    import _root_.java.awt._ ;
    import _root_.java.awt.event._ ;
    import _root_.javax.swing._ ;
    import _root_.javax.swing.event._;
""")
```

The *importHelper* object has import "macro-commands" to prepare ScalaLab for its main branches of functionality. Specifically:

Java standard UI and graphics support: *importJavaSwing()*

ScalaSci functionality based on JAMA library: *importScalaSciDefaultMat()*

ScalaSci functionality based on EJML library: *importScalaSciEJMLMat()*

ScalaSci functionality based on MTJ library: *importScalaSciMTJMat()*

ScalaSci functionality based on Apache Common Maths library: *importScalaSciCommonMathsMat()*

JMathPlot based standard plotting functionality: *importBasicPlots()*

However, since these import methods ("macro-commands") constructed by the *importHelper* object, call directly the main ScalaLab interpreter to perform the imports, their utility is available only within the ScalaLab environment. For example, the call *importJavaSwing()* in a stand-alone application will fail, since there is not a *GlobalValues.globalInterpreter* Scala interpreter object to evaluate the import statements. The solution for standalone applications is the direct injection of import statements described below.

Direct injection of import statements

The ScalaLab "Imports" menu option can be used to directly inject import statements concerning a specific library within the source code. This has the advantage of making the import statements

evident in the code, and also even more importantly the source code becomes **independent** of the ScalaLab environment and can be executed standalone. However, that comes at the expense of increasing the code size. For example, using the “NUMAL library imports directly”, the source is prepended with the code:

```
 import _root_.java.util.Vector ;
import _root_.numal._ ;
import _root_.numal.Algebraic_eval._;
import _root_.numal.Analytic_eval._
import _root_.numal.Analytic_problems._
import _root_.numal.Approximation._
import _root_.numal.Basic._;
import _root_.numal.FFT._;
import  _root_.numal.Linear_algebra._;
import _root_.numal.Special_functions._;
import java.text.DecimalFormat
```

Multiple complementary interfaces to the Scala Interpreter

This section describes how we can effectively work with multiple complementary interfaces to the ScalaInterpreter.

ScalaLab has three interfaces to work with the Scala Interpreter:

1. The **ScalaInterpreterPane** interface
2. The **RsyntaxArea** editor interface
3. The **ScalaLabConsole** interface that becomes visible only with the option “Show/Hide ScalaLabExplorer”, available from the “Configuration” menu

It is important top note that the ScalaInterpreterPane, the RSyntaxArea and the ScalaLabConsole interfaces use *the same* Scala Interpreter.

The **ScalaInterpreterPane** interface is the most convenient one and therefore is the default. We note, that the **RSyntaxArea** edit interface is similar, except that is based on a different editor.

The **ScalaLabConsole** interface fits well when we have to execute small one line scripts. The popup menu of the ScalaLabConsole has many useful operations, as to reset the Scala Interpreter or to create a new Scala Interpreter that uses different numerical libraries, e.g. the EJML (Efficient Java Matrix Library). These operations are also available and from ScalaLab's main menu options.

Let present some examples of working with these interfaces. Type the following piece of code within ScalaInterpreter pane:

```
var  a = ones0(20, 30)   // creates a zero-indexed 20X30 matrix filled with ones
var  b = ones0(30, 50)
var  c = a * b
```

We can execute this code either by selecting it and pressing **F6**, or line-by-line by pressing F6 at each line, or with CTRL-X, or with the execute pop-up menu option, or with **F2** that executes the text from the previously executed position up to the current cursor position.

Now we can go to the ScalaLabConsole window, and we can type the name of the matrix type *Mat* variable *a*. We observe that the contents of the matrix are printed, therefore the ScalaLabConsole shares the same Scala Interpreter with the ScalaInterpreterPane. This design decision allows to exploit effectively the facilities of both interfaces on the same data workspace.

Now let us define a new computation as:

```
var sc = size(c)

var rc = rand0(sc)
```

The code above takes the size of matrix c, and then creates a new random matrix of the same size. We can execute it at the ScalaLabConsole at this time.

Now we can return to the ScalaInterpreterPane, and we can type:

```
plot(rc(1, ::).getv)
```

Pressing F6 while we are at the same line, or by selecting the text and pressing F6 we can readily obtain a plot of the second line of the random matrix *rc*

Working with the Scala Interpreter Pane

The ScalaLab environment allows to execute easily ScalaLab code. We describe here the main points of working with the Scala Interpreter Pane editor.

Perhaps the most important keystroke is the **F6** or **'Ctrl-X',** or with the execute option of the pop-up menu, or **'Ctrl+Shift + E'** that permits to execute either:

- If the user has *selected text* in the editor, then the selected text is executed.
- Otherwise (i.e., no selected text) the *current line* is executed.

For example, if we have the line of code:

var alpha = 4.6; var beta = 8.9; var gg=alpha+9

Then to execute the whole line, we can press F6 with the cursor anywhere in the line. To execute only the second command concerning variable *beta*, we can select the command and press F6.

Similarly, the **F2** keystroke executes code from the previous F2 position (or start of text if no previous F2) up to cursor position.

The **F5** keystroke is also very useful, since it clears the output console, permitting to observe clearly the output messages and results.

The **F1** key collects information from the basic ScalaLab libraries in order to present code completion help for an identifier. Suppose for example that we want to acquire information for the *fft* routine in ScalaLab. We can select “fft” and press F1. Then a list is displayed with the ScalaLab classes that provide an *fft* routine (e.g. *JSci.maths.wavelet.Signal*). We can interrogate the contents of those classes by selecting the corresponding items from the displayed list and pressing a second F1. Then the contents of the class are retrieved using Java reflection and displayed. Also, the method of interest (e.g. *fft*) is highlighted.

The **F7** keystroke concerns also code completion and can be very useful when we want to retrieve the available contents (fields and methods) of an object. For example suppose we create a Java Swing JFrame with:

var jf = new javax.swing.JFrame("myFrame")

Then we can acquire the contents that start with *set* by typing:

jf.set and then F7

Similar is the **F4** keystroke but it uses Java reflection to construct the completion list. Also, the **Shift-F4** interrogates class contents, e.g. we can display the contents of *java.swing.JFrame* using a *JTree* based presentation.

Also the **F12** keystroke is useful when we have opened many figures and we want to close all of them.

ScalaLab by default presents on startup some comments about these keystrokes in order to help the user. The user can control the presentation of this help text using the *"Configuration"* top-level menu option, with the *"Toggle presentation of startup help"* submenu.

Compiling and using Scala code files

This section is concerned with Scala code kept in separate files, i.e. not Scala code scripts written within the ScalaLab's editor(s).

Since ScalaLab has access to the Scala compiler binary (i.e. file "*scala-compiler.jar* " in *the lib* folder), it can be used to compile any files containing Scala code with .scala extension as the *scalac* command does (actually this command uses the same compiler). In this section we will show by means of an example how to compile and access code defined as a normal Scala source file.

Suppose that we have code placed within a Scala object that implements a mathematical function, for simplicity we consider the simple *cube* function implemented with the *CubeObj* Scala object, which is in a file named *simpleMaths.scala*. Let the code in *simpleMaths.scala* be the following:

```
object CubeObj {

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

}
```

Let place *simpleMaths.scala* within a folder, say for example, */home/sp/testScalaLab* in my Linux workstation.

In order to access the code of simpleMaths.scala, the first step is to *compile* the Scala source. We can use either the ScalaLab explorer or the usual *scalac* command for that task. The later option clearly requires an installation of Scala. One way to achieve that task with the ScalaLabExplorer, is

to type the path */home/sp/testScalaLab* in the text edit control with label "Specify Path", then pressing "Browse", displays the contents of the folder. By selecting the file *simpleMaths.scala*, right-clicking the mouse, and selecting "Scala -> Compile .scala file" we can compile the code directly. ScalaLab uses its integrated Scala compiler and does not require an installation of Scala.

The second step, is to append the directory */home/sp/testScalaLab* to the classpath of the Scala interpreter (i. e. to the *ScalaLabClassPath*). One way to achieve this is to select the path, right-mouse-click and then select from the popup menu "Paths->Append the path to the ScalaSci Paths". Subsequently, we need a new Scala Interpreter that includes that path on its classpath. By right-mouse clicking over the *ScalaLabConsole* area, we can select "Reset Scala Interpreter using ScalaLab default imports" option. A new Scala Interpreter is created and as can be verified from the contents of the classpath that are displayed, it includes the new path. Also, we can instantiate a new Scala interpreter from the “Scala Interpreter” menu item from the main menu bar of ScalaLab.

Finally, we can utilize the functionality of the new compiled code from the Interpreter, i.e. we can compute the cube of a number and apply our cube function to the elements of a *Vec* object:

```
// compute the cube of a number

val anum = 5.6

val anumCube = CubeObj.cube(anum)

println("cube of "+anum+" = "+anumCube)

// apply the function to the elements of a vector

val t = linspace(0, 5, 2000)

val x = t map CubeObj.cube

plot(t,x)
```

We should note, that by exiting ScalaLab, the paths configuration is saved and is available at the next session. We can remove paths, using the JTree control that displays user defined class paths, and by pressing the DEL key over a corresponding tree node. The node is deleted from the ScalaSci classpaths, but we should create a new Scala interpreter before exiting, because it is the classpath of the running Scala interpreter that is saved upon exiting.

Controlling verbose of output for matrices/vectors, digits of precision, truncation of large matrices/vectors

It is very convenient practically to control the output of commands. For that reason ScalaLab implements a menu option at the Configuration menu, i.e. the "Control the format of displayed numbers and truncation of large matrices" that also controls the verbose on/off status. The preferences selected by the user are saved and are available for the next ScalaLab session.

Also, it is easy to access these controls programmatically through the following ScalaSci object:

```
package ScalaSci
//  parameters that determine the format with which numbers and arrays are printed
object PrintFormatParams {
   var verboseFlag = true   // controls verbosing the results of  toString()
   def setVerbose(vflag: Boolean) = { verboseFlag = vflag }
   def getVerbose() = verboseFlag

   var vecDigitsPrecision = 4   // controls precision of toString()
   def getVecDigitsPrecision() = vecDigitsPrecision
   var vecMxElemsToDisplay = 20    // controls maximum number of Vector elements to display
   def setVecMxElemsToDisplay( mxElems: Int): Int  = { var prevElems = vecMxElemsToDisplay;
vecMxElemsToDisplay = mxElems; prevElems}
   def setVecDigitsPrecision(precision: Int) = { vecDigitsPrecision = precision }

    var matDigitsPrecision = 4  // controls precision of toString()
    def getMatDigitsPrecision() = matDigitsPrecision
    var matMxRowsToDisplay = 6
    var matMxColsToDisplay = 6
    def getMatMxRowsToDisplay() = matMxRowsToDisplay
    def getMatMxColsToDisplay() = matMxColsToDisplay
    def setMatMxRowsToDisplay(nrows: Int) = { matMxRowsToDisplay = nrows }
    def setMatMxColsToDisplay(ncols: Int) = { matMxColsToDisplay = ncols }
    def setMatDigitsPrecision(precision: Int) = { matDigitsPrecision = precision; vecDigitsPrecision =
precision }
}
```

Additionally, the same routines are available at the REPL mode of ScalaLab operation, with

commands available through the *BasicCommands* object. These commands are (also displayed with the command "help"):

def setVerbose(vflag: Boolean) // controls verbosing the results of toString()

def getVerbose() // returns the current verbosing state

def setVecMxElemsToDisplay(mxElems: Int) : Int // controls the number of elements displayed for vectors

def setVecDigitsPrecision(precision: Int) // controls the precision of vector elements

def getVecDigitsPrecision() // returns the precision with which vector elements are displayed

def getMatMxRowsToDisplay() // returns the number of rows displayed for matrices

def getMatMxColsToDisplay() // returns the number of cols displayed for matrices

def setMatMxRowsToDisplay(nrows: Int) // sets the number of rows displayed for matrices

def setMatMxColsToDisplay(ncols: Int) //sets the number of cols displayed for matrices

def setMatDigitsPrecision(precision: Int) // controls the precision of Matrix elements

def getMatDigitsPrecision() // returns the precision with which Matrix elements are displayed

Building ScalaLab with sbt

To build ScalaLab with **sbt** is very easy.

Specifically, we can download the .zip file of ScalaLab from github. It contains both the sources and all the relevant libraries to build ScalaLab with sbt.

Then we run the script *runSbt.sh* that starts sbt configuring it to use a larger heap size.

At the sbt shell prompt:

1. we type ***sbt*** from the folder where *build.sbt* is unzipped
2. then we type ***clean***

and

3. finally the sbt's command ***package*** builds the ScalaLab's executable .jar file**.**

Building ScalaLab with ant

The Apache's ant is used internally by the Netbeans environment configured with the Scala's plugin. However, we can use ant externally to build ScalaLab, although sbt is the preferred way.

To build ScalaLab from sources with **ant** is very simple.

Download the .zip file of ScalaLab from github. It contains both the sources and all the relevant libraries to build ScalaLab with ant.

To build ScalaLab is very simple:

1. Unzip the zip file

2. Go to .\ScalaLabPr folder and build with ant, i.e.

ant

The executable is built in the dist folder

Chapter 2 Basic Mathematical Classes

The basic Matrix-like classes of ScalaLab

ScalaLab has many matrix like classes in order to present different functionality and to interface with different matrix libraries. Therefore it is useful to follow and learn some basic conventions in order to be able to work effectively without confusion. These conventions are described below.

RichDouble2DArray: The *RichDouble2DArray class* wraps the Java/Scala 2-D double arrays and is one of the more important and effective matrix type in ScalaLab. It does not concentrate on a specific library, instead it aims to **exploit the best algorithms in its numerical operations.** Since it is designed to be the most effective type, the static routines are in their direct format (e.g. *ones(), zeros(), rand(),* i.e. routines without a trailing "1" or "0", as e.g. *rand())* return a *RichDouble2DArray* object, for example:

var rra = rand(9, 10) // returns a RichDouble2DArray object

Zero-indexed Matrix Libraries

These libraries use a matrix that is indexed starting with the zero position, i.e. as Java or C/C++ arrays, but not as Fortran or MATLAB arrays that start at index 1.

There are several zero-indexed matrix classes in ScalaLab, each of which wraps the functionality of important Java numerical libraries. This set of libraries is extendable and more libraries can be easily adapted. Currently, the important libraries have as follows:

The *ScalaSci.Mat* class wraps the JAMA based routines, *ScalaSci.EJML.Mat* the EJML library, *ScalaSci.MTJ.Mat* the MTJ library and *ScalaSci.CommonMaths.Mat* the Apache Common Maths library. In order not to have ambiguities, we initialize a different Scala Interpreter for each such library. Whatever is the matrix library that the Scala Interpreter targets, static routines that end in 0 (zero) cope with the currently utilized library wrapper object. For example:

var mra0 = rand0(2,3) // returns a random zero indexed Mat depending on the targeted library by the Scala Interpreter

// e.g. ScalaSci.EJML.Mat for the EJML Interpreter, ScalaSci.MTJ.Mat for the MTJ Interpreter etc.

One-indexed Matrix Library

The one-indexed Matrix library wraps the NUMAL library. By convention routines with a trailing 1 (e.g. rand1()) operate with that Matrixes, e.g.

var mra1 = rand1(2,3)

RichDouble1DArray This ScalaSci class wraps the standard Java/Scala 1-D arrays of doubles. It has overlapping functionality with the *ScalaSci.Vec* class. Static routines accepting one dimension argument, return such an object, e.g.

var r1d = rand(9) // RichDouble1DArray

Array[Array[Double]] The standard Java-Scala 2-D double arrays. We have also to capitalize the first letter at the static routines, to have such an object, i.e.

var r2D = Rand(3,7) // Array[Array[Double]

Vectors The *ScalaSci.Vec* implements a Vector style of class. Static routines returning such objects are prefixed with a letter 'v', e.g.

var vrnd = vrand(9) // vector

Important Conventions for operating effectively with different matrix types

There are some important conventions about the naming of static routines, as for example: *rand(n,m)* copes with a *RichDouble2DArray* type, *rand1(n, m)* with an one-indexed NUMAL based *ScalaSci.Matrix* type.

These conventions are very simple and are illustrated with the code below:

```
// test rand()
var  mra1 = rand1(2,3)   //  one indexed Matrix
var  mra0 = rand0(2,3)   //  zero indexed Mat
var rra = rand(2,3)  //  RichDouble2DArray
var r1D = Rand(9) //  Array[Double]
var r2D = Rand(3,7) // Array[Array[Double]
var vrnd = vrand(9)  // ScalaSci.Vec (vector  type)
```

```
// test fill()
var v = 4.5
var  mfa1 = fill1(2,3, v)   //  one indexed Matrix
var  mfa0 = fill0(2,3, v)   //  zero indexed Mat
var mf1d = fill(6, v) // RichDouble2DArray
var rfa = fill(2,3, v)  //  RichDouble2DArray
var v1D = Fill(9, v)  // Array[Double]
var v2D = Fill(3,9, v) // Array[Array[Double]]
var vf = vfill(7, v)  // vector
// test ones()
var  moa1 = ones1(2,3)   //  one indexed Matrix
var  moa0 = ones0(2,3)   //  zero indexed Mat
var  mo1d = ones(8)  // RichDouble2DArray
var mo2d = ones(2,3)  //  RichDouble2DArray
var mO1D = Ones(8)  // Array[Double]
var mO2D = Ones(6,8)  // Array[Array[Double]]
var vo = vones(9)  // vector
// test zeros()
var  mza1 = zeros1(2,3)   //  one indexed Matrix
var  mza0 = zeros0(2,3)   //  zero indexed Mat
var  mz1d = zeros(8)  // RichDouble2DArray
var mz2d = zeros(2,3)  //  RichDouble2DArray
var mZ1D = Zeros(8)  // Array[Double]
var mZ2D = Zeros(6,8)  // Array[Array[Double]]
var vz = vzeros(9)  // vector
```

Description of the RichDouble2DArray class

The RichDouble2DArray wraps standard Java/Scala 2-D Arrays of doubles.

Constructors

RichDouble2DArray**(rows: Integer, cols: Integer)** // Creates a RichDouble2DArray of size rows, cols initialized to zeros

var m=new RichDouble2DArray(2, 3)

Construction by specifying the initial elements, e.g.

var m = AAD("4.5 5.6 -4; 4.5 3 -3.4") // space separated

var m = AAD("4.5, 5.6, -4; 4.5, 3, -3.4") // comma separated

Another type of construction:

```
var xx = 3.4
var a = RD2D( 2, 4,  // specify the size first
  3.4, 5.6, -6.7, -xx,
  -6.1,  2.4, -0.5, cos(0.45*xx))
```

// construct RichDouble2DArray by copying the array values

RichDouble2DArray(da: Array[Array[Double]]) // Creates a RichDouble2DArray initialized with the da array

var dd = Array.ofDim[Double](2,4)

dd(1)(1)=11

var mdd = new RichDouble2DArray(dd)

Basic Methods

def size: (Int, Int) // Returns the number of rows and columns of the RichDouble2DArray

def numRows(): Int // Returns the number of rows of RichDouble2DArray

def numColumns(): Int // Returns the number of columns of RichDouble2DArray

def length: Int // Returns the number of rows of RichDouble2DArray

Conversion Routines

The RichDouble2DArray class aims to be the main Matrix type of ScalaLab and very convenient to use. Therefore routines are provided to convert from other matrix types. Some conversions are:

Convert from a JAMA based matrix (i.e. ScalaSci.Mat)

var a = ScalaSci.Mat.rand(2, 5) // a JAMA based matrix

var ra = new RichDouble2DArray(a) // convert to RichDouble2DArray

Convert from an NUMAL based matrix (i.e. ScalaSci.Matrix)

var a = ScalaSci.Matrix.rand(2, 5) // a NUMAL based matrix

var ra = new RichDouble2DArray(a) // convert to RichDouble2DArray

Convert from an EJML based matrix (i.e. ScalaSci.EJML.Mat)

var a = ScalaSci.EJML.StaticMathsEJML.rand0(2, 5) // an EJML based matrix

var ra = new RichDouble2DArray(a) // convert to RichDouble2DArray

Convert from an MTJ based matrix (i.e. ScalaSci.MTJ.Mat)

var a = ScalaSci.MTJ.StaticMathsMTJ.rand0(2, 5) // an MTJ based matrix

var ra = new RichDouble2DArray(a) // convert to RichDouble2DArray

Convert from a Common Maths based matrix (i.e. ScalaSci.CommonMaths.Mat)

var a = ScalaSci.CommonMaths.StaticMathsCommonMaths.rand0(2, 5) // a Common Maths based matrix

var ra = new RichDouble2DArray(a) // convert to RichDouble2DArray

Routines

***def ones(n: Int, m: Int): RichDouble2DArray** - constructs a nXm RichDouble2DArray of ones*

***def zeros(n: Int, m: Int): RichDouble2DArray** - constructs a nXm RichDouble2DArray of zeros*

***def rand(n: Int, m: Int): RichDouble2DArray** - constructs a nXm RichDouble2DArray of random values*

*d**ef eye(n: Int, m: Int): RichDouble2DArray** - constructs a nXm RichDouble2DArray of zero everywhere except the diagonal*

Transpose operator ~

transposes the RichDouble2DArray , e.g.

```
var   rda = rand(9,12)  // create RichDouble2DArray of size 9 X 14
var   rdat = rda~   // transpose it
```

2-D Arrays – The enhanced ScalaLab Array[Array[Double]] type

Scientific libraries usually design a “matrix” class and build mathematical operations around it. Although, ScalaLab has such matrix classes (and most of them serve as wrappers around matrix classes of Java libraries), much convenience in expressing mathematical expressions is built around the standard two-dimensional double arrays. This allows to work effectively with simple Java like arrays and also to exploit numerical algorithms implemented in different numerical libraries.

ScalaLab extends significantly the potential of performing mathematical operations with the standard Java/Scala 2-D double arrays.

Creating arrays initialized with values

We can create 1-D arrays with pre-specified elements either as with Scala, e.g.

```
var  aa = Array(7.8, -9.8, 8.9)
```

or as:

```
var  aa = AD("7.8, -9.8, 8.9")
```

Similarly for 2-D Arrays:

```
var  aa2 = Array(Array(7.8, -9.8, 8.9), Array(2.3, -4.5, -44.5))
```

or as:

```
var  aa2 = AAD("7.8, -9.8, 8.9; 2.3 -4.5 -44.5")
```

The elements in the *Array()* factory method can be either comma or space separated and a semicolon (';') introduces new rows.

Operations on arrays

Let create a 2-D array:

var A = Rand(8, 12)

We define a function that adds a constant 5 to a number.

def inc5(x: Double) = x+5

We can directly apply a function to all the elements of a 2-D double array with the *map* function:

var A5 = A map inc5 // map a function

We negate the array simply as:

var A5m = -A5 // unary minus

We multiply all the elements with 10.

*var A10 = 10*A*

The operation above works by converting 10 to a *RichNumber* and then the multiplication with the Array[Array[Double]] is performed. In the same spirit, below A is converted to a *RichDouble2DArray* and then multiplication is performed:

*var A20 = A*20.0*

We can directly apply a lot of mathematical functions, e.g.: *sin(A), cos(A), tan(A), cosh(A), sinh(A), tanh(A), log(A), exp(A).*

The *transpose* of the array can be taken as:

var At = A~

We can multiply arrays as usual:

*var aa = A*At*

We can express convenient expressions, e.g.:

var ca = A-2+7(A+3*A)-cos(A)+tan(A-8+0.3*A)*

The RichDouble1DArray (1-D Double Arrays)

The *RichDouble1DArray* object actually wraps an Array[Double] of Scala or a double[] of Java. However, it

presents a large number of *static methods* to the interpreter in order to facilitate the work of the user. Such methods allow more convenient handling with overloading the basic mathematical operators, and using the basic mathematical functions (e.g. sin, cos, tan, etc.) directly on RichDouble1DArray objects.

We present these operations by means of examples.

Operations on 1-D Double Arrays

Construction from elements:

*var ad = **AD**("3.4 -6.7 -1.2 5. 6")*

or

*var ad1 = **RD1D**(3.4, -6.7, -1.2, 5.6)*

Construction from elements 1-indexed array:

*var ad1 = **AD1**("3.4 -6.7 -1.2 5. 6")*

Variance of an array:

*var av = **Var**(ad)*

Standard Deviation:

*var astd = **std**(ad)*

Covariance:

*var acov = **covariance**(ad, ad)*

Correlation:

*var acor = **correlation**(ad, 6*ad)*

The objective of the RichDouble1DArray class is to provide convenient operations on Java/Scala 1-D arrays of doubles. Static operations like for example *sin*, *cos*, *tan*, *log*, *ceil* etc are provided with the *RichDouble1DArray* class. These operations convert implicitly the *Array[Double]* type, for example:

var x = new Array[Double](20)

x(2) = 6

var y = sin(x) // sin is provided with the RichDouble1DArray class, and returns also an Array[Double] object

However, we cannot provide convenient operations, like addition, multiplication etc. on an Array[Double] object, since it is a predefined type. Therefore, we exploit the implicit conversions machinery of Scala by

means of the conversion to the RichDouble1DArray type. For example:

var x = new Array[Double](20)

x(2) = 6

var y = sin(x)+3 // sin is provided with the RichDouble1DArray class, and returns also an Array[Double] object

The RichDouble2DArray

The RichDouble2DArray class also provides operations on two-dimensional array of doubles, corresponding to the Scala type Array[Array[Double]], and interoperable with the double[][] type of Java. A large number of operations is supported on this class with a syntax similar to the Matrix class.

The Vec class

The **Vec** class implements one-dimensional dense vectors in ScalaSci.

Constructors

1. **Vec(len: Integer): Creates a vector of size len initialized to zeros**

var v = new Vec(100) // creates a vector with 100 elements initialized to zero values.

2. Vec(da: Array[Double]): Creates a vector initialized with the da array

var da = new Array[Double](20)

da(3)=3

var dav = new Vec(da)

3. **Construction by specifying the initial elements, e.g.**

   var v = V("4.5 5.6 -4") // space separated

   var vv = V("4, -5.6, 6.7, -100.1") // comma separated

Basic Methods

def size: Int // Returns the size of the Vector

def length: Int // Returns the size of the Vector

var l = dav.size

Displaying the whole vector contents

When displaying the vector contents, ScalaLab calls the *toString()* method of the *Vec* class, that displays the first values only of large vectors.

The whole contents of the vector are printed to Console Standard Output with the **print** routine, e.g.

var v = vrand(40) // a random vector with 40 elements

v.print

In-Place operators

In-place operators aim for efficiency. They do not create a new object, but instead they perform the operation by mutating the object on which they are applied. Although this violates the principles of functional programming, it can offer significant performance benefits, of about 3 to 6 times speed increase in loops involving a lot of vector operations, since the garbage collection overhead is avoided. For example such in-place operators are the ++, –, ** operators, demonstrated at the example below:

var v = vrand(100)

var vo = vones(v.length)

v++vo // adds to v the vo vector in place, i.e. the results replace the previous contents of v,

// it is as the operation v=v+vo, but more efficient

*var v10 = 10*vones(v.length)*

v--v10 // subtract in-place

*v**100 // multiply in-place with 100*

Scalar append Vector operator ::

Appends a scalar to the end of a vector, e.g.

var vo = vones(4)

*var vo5 = 5 :: vo // the operation is **mutable,** i.e. the vector vo also is changed*

Scalar prepend Vector operator :::

Prepends a scalar to the start of a vector, e.g.

var vo = vones(4)

*var vo6 = 6 ::: vo // the operation is **mutable**, i.e. the vector vo also is changed*

Vector append Scalar operator ::<
Appends a scalar to the end of a vector, e.g.

var vo = vones(4)
*var vo5 = vo ::< 5 // the operation is **mutable**, i.e. the vector vo also is changed*

Vector prepend Scalar operator :::<
prepends a scalar to the start of a vector, e.g.

var vo = vones(4)
*var vo6 = vo :::< 6 // the operation is **mutable**, i.e. the vector vo also is changed*

Vector append Vector operator :::
appends the first vector at the start of the second, e.g.

vones(2) ::: vrand(5)

inc(x1, dx, x2) or x1::dx::x2 : implements colon operator of MATLAB, i.e. x1:dx:x2
var (x1, dx, x2) = (-12, 0.01, 18)
var x = inc(x1, dx, x2)
*var y = sin(0.12*x)+2.3*cos(1.23*x)*
plot(x,y)

linspace: Linearly spaced vector.

linspace(x1, x2) generates a row vector of 100 linearly equally spaced points between x1 and x2.
linspace(x1, x2, N) generates N points between x1 and x2. For example:

var N = pow(2, 14).asInstanceOf[Int] // signal length
var t = linspace(0, 10, N) // sample interval [0, 10] with 2^12 equally spaced points
var F1=30; var F2 = 4; // signal frequencies
*var x = sin(F1*t)+2.4*cos(F2*t) // construct synthetic signal*
var fftx = afft(x) // perform FFT using Apache Common Maths

def getReal(x: org.apache.commons.math3.complex.Complex) = x.getReal
var fftxReals = fftx map getReal

linePlotsOn // connect the points with a line (scatter plots draw them as they are, i.e. unconnected)

figure(1); subplot(2,1,1); plot(x); title("the signal");

subplot(2,1,2); plot(fftxReals); title("FFT of the signal - real parts")

logspace: Logarithmically spaced vector.

logspace(x1, x2, N, base) generates a row vector of N logarithmically equally spaced points between x1 and x2.

e.g.

var x = logspace(1, 4, 100, 10)

var N = pow(2, 14).toInt // signal length

var tdefault = logspace(0,10, N) // default base 10

var base = 2

var t = logspace(0, 10, N, base) // sample time axis

base = 10

var t10 = logspace(0, 10, N, base) // sample time axis

dot : Vector dot product, e.g.

var rv1 = vrand(1000) // a random vector

rv1 = rv1-mean(rv1) // mean subtract

var rv2 = vrand(1000) // another random vector

rv2 = rv2-mean(rv2)

var rv1dprv2 = rv1 dot rv2 // should be small, since vectors are uncorrelated

var rv1dprv1 = rv1 dot rv1 // should be large since is the dot product of a vector by itself

vones(N: Int) : *generates a N-sized vector of ones*

vzeros(N: Int) : *generates a N-sized vector of zeros*

vfill(N: Int, value: Double): *generates a N-sized vector filled with value*

abs, min, max, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, ceil, floor, sqrt, log, exp: *the common mathematical routines implemented on vectors*

sum(v: Vec) *: sums the element of the vector*

mean(v: Vec) : *returns the mean value of the elements of the vector*or

vrand(N: Int): returns a uniformly distributed random vector of size N

var v = vrand(2000)

plot(v)

title("2000 random values")

vones(N: Int): returns a N-sized vector of 1s

var v = vones(300)

vzeros(N: Int): returns a N-sized vector of 0s

var v = vzeros(300)

sum(v: Vec): sums the elements of v

var v = vones(20)

var sm = sum(v)

mean(v: Vec): the mean of the elements of v

var v = vrand(200)

var mn = mean(v)

vfill(N: Int, value: Double): returns a N-sized vector filled with value

var v = vfill(300, -0.24)

sin, cos, tan, asin, acos, asin, sinh, cosh, tanh: The corresponding trigonometric routines operating on Vectors

var t=inc(-0.9, 0.01, 0.9)

*var x = tanh(4*t)*

figure(1); subplot(3,2,1); plot(t,x);

var xasin = asin(t); subplot(3,2,2); plot(t, xasin);

var xacos = acos(t); subplot(3,2,3); plot(t, xacos);

var xsinh = sinh(t); subplot(3,2,4); plot(t, xsinh);

var xcosh = cosh(t); subplot(3,2,5); plot(t, xcosh);

*var xcomp = 4*cos(4.5*t)+0.34*sin(-18.364*t); subplot(3,2,6); plot(t, xcomp);*

abs, ceil, floor, sqrt, log, exp: The corresponding simple routines operating directly on Vectors

fft: Compute Fast Fourier Transform

var F1=30; var F2 = 4; // signal frequencies

var t = linspace(0, 10, 2^^^12)

*var x = sin(F1*t)+2.4*cos(F2*t) // construct synthetic signal*

var fftx = fft(x) // perform FFT

linePlotsOn

figure(1); subplot(3,1,1); plot(x); title("the signal");

subplot(3,1,2); plot(fftx._1); title("FFT of the signal - real parts")

subplot(3,1,3); plot(fftx._2); title("FFT of the signal - imaginary parts")

Matrix Select – Update Operations

It is convenient to be able to select/update Matrix ranges with a MATLAB-like style. ScalaLab implements such routines that select and update Matrix sub-ranges. These routines have the same interface independent of the Matrix type. We illustrate those operations by means of examples.

Examples of Matrix range select operations

Let create a zero-indexed matrix and fill it with some entries.

var N = 40; var M = 50;

var a = new Mat(N,M)

for (r<-0 until N)

for (c<-0 until M)

*a(r, c) = r*c*

We can select the row with index 2 with:

val ridx = 2

val ar2 = a(ridx, ::)

We can select rows 1 and 2 as:

val a_r1_r2 = a(1, 2, ::)

or

val a_r1_r2 = a(1::2, ::)

We can specify a "step" parameter, in order not to take rows consecutively, e.g.

val a_r1_s2_r2 = a(1::2::8, ::)

Symmetrically we can work with columns:

val cidx = 3

val ac3 = a(::, cidx)

val a_c1_c3 = a(::, 1::3)

val a_c1_s2_c5 = a(::, 1::2::4)

We can select across both rows and columns using the parameter pattern *a(startRow, stepRow, endRow, startCol, stepCol, endCol)* , e.g.

val apart = a(0, 2, 3, 0, 2, 4)

or

val apart = a(0::2::3, 0::2::4)

Examples of Matrix range update operations

Let create two simple zero-indexed matrices

var a = rand0(4, 8)

var o = ones0(2, 3)

We can update a matrix range copying another matrix at the specified index as:

a(1, 2, ::)= o // copy matrix o within a starting at (1,2)

The operation above overwrites the previous matrix contents at the corresponding positions.

We can also specify steps for rows and columns, in order not to copy the source matrix consecutively, e.g.

var al = rand0(15, 30)

var ol = ones0(12, 13)

var dx = 2; var dy = 3

al(1, 2, dx, dy)= ol

We can use a vector to fill Matrix rows or columns. The use of a vector in order to fill rows and columns of a matrix can be convenient. Here are some examples:

val a = zeros0(10, 50)

val v50 = vfill(50, -1) // used to fill columns

val v10 = vfill(10, 1) // used to fill rows

a(2, ::) = v50 // row 2 has -1 s

a(::, 3) = v10 // column 3 has 1s

Some more examples of matrix select/update operations are illustrated with the following script:

// create a random matrix

var xx = rand(20)

var rowS = 1; var rowE = 16; var rowI = 2

var colS = 1; var colE = 16; var colI = 2

// select a row range

var ff = xx(rowS::rowE, ::)

// update that row range

xx(rowS::rowE, ::) = 8.8

// select a row range with increment rowI

var fff = xx(rowS::rowI::rowE, ::)

// update that row range

xx(rowS::rowI::rowE, ::) = 55.4

// select a column range

var ffc = xx(::, colS::colE)

// update that column range

xx(::, colS::colE)= -99.9

// select a column range with increment colI

var fffc = xx(::, colS::colI::colE)

// update that column range

xx(::, colS::colI::colE)= 33

// some more select operations

var sel1 = xx(rowS::2::rowE, 3::2::colE)+5.7

var sel2 = xx(rowS::rowE, 3::2::colE)+57.9

var sel3 = xx(rowS::rowE, 3::colE)

var sel4 = xx(rowS::rowE, 3::2::colE)

var sel5 = xx(rowS::rowE, ::)

var sel6 = xx(::, 3::colE)

var sel7 = xx(2, ::) // 2nd row

var sel8 = xx(::, 2) // 2nd column

var sel9 = xx(2, 2::3) // 2nd row, elements 2 up to 3

var sel10 = xx(2, 2::2::10) // 2nd row, elements 2 up to 10 by step 2

var sel11 = xx(3::6, 3) // 3nd column, rows 3 up to 6

var sel12 = xx(2::3::15, 4) // 4th column, rows 2 to 15 by step 3

// And some update operations:

var x = rand0(20); x(1::2::6, 2::3)= 150.4

var x2 = rand0(20); x2(3::5, 2::2::12)= 50.4

var x3 = rand0(9); x3(2::3, 5::6)=88

x(2,::) = 22.3 // all elements of row 2 to 22.3

x(::, 3) = 12 // all elements of column 3 to 12

x(2, 3::4) = 44 // elements 3 up to 4 of row 2 to 44

x(2, 3::2::12) = 122.2 // elements 3 up to 12 by step 2 of row 2 to 122.2

x(::, 3) = 33.3 // all elements of column 3 to 33.3

x(2::5, 3) = -4.3 // elements of rows 2 up to 5 of column 3 to -4.3

x(2::3::15, 3) = sin(x(1,1)) // elements of rows 2 to 15 by step 3 to sin(x(1,1))

Consistency Across Many Matrix Libraries

An important advantage of ScalaLab is that it can utilize effectively and flexibly different Java/Scala Matrix libraries. Each such library has its own data representations with the consequent advantages/disadvantages and implements its own numerical routines.

However, at the original ScalaSci design, the complexity and non-uniformity of the Matrix libraries, resulted in an inconsistent ScalaSci interface of the Matrix routines. Therefore, we redesigned ScalaSci in order to utilize Scala features that enforce the consistency with the help of the Scala compiler. Below we present those design features.

Many important classes of ScalaSci, such as the *Vec* , the one-indexed matrix class *Matrix* and the highly convenient *RichDouble2DArray* class that acts as the default matrix type are independent of the particular zero-indexed Matrix library. Therefore, these routines are always available, independently of the particular zero-indexed library at which the Scala Interpreter switches. The ScalaLab user can switch the zero-indezed matrix support library (e.g. from EJML to Apache Common Maths), and still use the same code.

In order to abstract the functionality that each ScalaSci matrix class should implement we have designed two traits, the ***ScalaSciMatrix*** that defines the mandatory functionality from each Matrix class and the ***StaticScalaSciGlobal*** trait, that defines also the mandatory functionality of the static routines that correspond to the currently utilized zero-indexed Matrix class.

Each ScalaSci matrix wrapper class mixes-in the ScalaSciMatrix trait. We should note also that this trait defines a large number of **in-place** operations, e.g.

var a = ones0(20, 30)

*a.sin // take the sine of all matrix elements,**in-place, i.e.** without creating another matrix*

These operations yield efficiency, especially for large matrices, by avoiding extensive memory allocation and garbage collection operations.

Also, each ScalaSci Static Imports Object mixes in the *StaticScalaSciCommonOps* trait, in order to force the implementation of the common API.

Below we describe in some detail the ***ScalaSciMatrix*** trait because it provides many useful operations that are available for all the Matrix types of ScalaLab.

The ScalaSciMatrix trait

The *ScalaSciMatrix* trait defines and enforces the basic functionality of all ScalaSci Matrix types. It also implements the common patterns of functionality for all matrix types. The user should use these routines in order to have portable ScalaSci code, that can directly use different lower-level matrix libraries.

We describe some important operations provided by the *ScalaSciMatrix* trait to any *ScalaSci* matrix type.

Let create a 5X5 random matrix:

var x = rand0(5, 5)

We can get the number of its rows and columns with (either as public fields or with method calls):

var nr = x.numRows

var nc = x.numColumns

var nr2 = x.numRows()

var nc2 = x.numColumns()

The *getv()* routine returns *the low-level data representation of the Matrix*, whatever it is. Common representations are for example, a *one-dimensional array of doubles* (|i.e. double[]) arranged in either *row-storage* format or *column-storage* format, or a two-dimensional array of doubles (i.e. double [][]). For example, for the EJML library *getv()* returns a one-dimensional array of doubles, since EJML works by mapping two-dimensional matrices to a one-dimensional Java array of doubles. We can call *getv* as:

var xv = x.getv

The *getLibraryMatrixRef()* routine is also very useful. It returns *the library dependent class* that implements native operations of the library (which is usually a Java library and not C/C++ “native” code, although it can be the later). This allows ScalaLab code to combine Scala implemented operations, with the existing operations provided by the Java library. In this way the full potential of the underlying Java class can be utilized. For example for an EJML based matrix type:

var libEJML = x.getLibraryMatrixRef

returns the *org.ejml.simple.SimpleMatrix* on which the ScalaLab EJML-based matrix is based.

The *matFromLibrary()* method constructs the ScalaSci matrix from the library representation whatever the library representation is. In that way we can process the native representation with the methods of the native library and after that take the updated matrix class. Clearly, this method can be combined with *getLibraryMatrixRef()* to switch back and forth between the library and ScalaSci matrix object formats.

The *ScalaSciMatrix* trait also provides many *apply()* and *update()* methods for convenient matrix indexing and update, and a lot of filtering and utility methods.

Additionally, some very useful routines are defined with the ScalaSciMatrix trait, as for example:

```
// det() is the determinant of the square matrix X.
def det(): Double

/* trace(): is the sum of the diagonal elements of A, which is
   also the sum of the eigenvalues of A. */
def trace(): Double

def inv(): specificMatrix // the inverse of the matrix

def pinv(): specificMatrix // the pseudo-inverse of the matrix

 /*returns the 2-norm condition number (the ratio of the
   largest singular value to the smallest). Large condition
   numbers indicate a nearly singular matrix.
 */
def cond(): Double
// provides an estimate of the number of linearly independent rows or columns of a matrix A.
def rank(): Int

/* (V,D) = eig(X) : produces an array D of eigenvalues and a
```

full matrix V whose columns are the corresponding eigenvectors so

*that X*V = V*D */*

def eig(): (RichDouble2DArray, RichDouble1DArray)

/ (U,S,V) = svd(X) produces a diagonal matrix S, of the same*

dimension as X and with nonnegative diagonal elements in

decreasing order, and unitary matrices U and V so that

*X = U*S*V'.*

For example:

var x = rand(N,N)

var (u,s,v) = svd(x)

*var shouldBeZero=u*diag(s)*(v~)-x*

/

def svd(): (RichDouble2DArray, RichDouble1DArray, RichDouble2DArray)

}

NUMAL library interface and the one-indexed Matrix class

Another basic library is the NUMAL Java library described in [11]. 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]]* (i.e. Java's double [][]) type and they are imported by default in the Scala Interpreter. Also, the *RichDouble2DArray* 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.

The **Matrix** class in ScalaLab implements a 1-indexed matrix class, i.e. if **a** is a variable of type Matrix the first element is accessed as a(1,1) instead of the C/C++/Java like zero-based indexing of a(0,0). This type of indexing is convenient for interfacing with the NUMAL library which adopts one-based indexing of arrays. We note that MATLAB and Fortran also, are two heavily used languages in scientific computing that use one-indexing.

Therefore, the **Matrix** class implements one-indexed two-dimensional dense matrices in ScalaSci that are very useful for using the powerful **NUMAL** Numerical Library supplied with the book:

"A Numerical Library in Java for Scientists & Engineers", Hang T. Lau, Chapman & Hall/CRC, 2004

Constructors

Matrix(rows: Integer, cols: Integer) // Creates a Matrix of size rows, cols initialized to zeros

```
var m=new Matrix(2, 3)
```

Construction by specifying the initial elements, e.g.

```
var m1 = M1("4.5 5.6 -4; 4.5 3 -3.4") // space separated
var m2 = M1("4.5, 5.6, -4; 4.5, 3, -3.4") // comma separated
```

// construct Matrix by copying the array values
Matrix(da: Array[Array[Double]]) // Creates a Matrix initialized with the da array

```
var dd = new Array[Array[Double]](2,4)
dd(1)(1)=11
var mdd = new Matrix(dd)
```

// construct a Matrix by **reference an existing double array**,
// in order to avoid duplicating the array

```
var tm = new Matrix(1,1) // a small Matrix object
tm.setv(dd, 1, 3) // set its value content to an existing double array of size 2X4
```

Matrix(rows: Integer, cols: Integer, initValue: Double) // Creates a Matrix of size rows, cols initialized to initValue

```
var m=new Matrix(3, 4, 3.4)
```

// construct a Matrix from a two-dimensional double array without copying the array values

Matrix(da: Array[Array[Double]], flag: Boolean) // Creates a Matrix initialized with a reference to the da array

// construct a Matrix from an one-dimensional double array

Matrix(da: Array[Double])

var od = new Array[Double](10)

od(2)=2.2

var odm = new Matrix(od)

Basic Methods

def size: Int // Returns the number of rows and columns of the Mat

def length: Int // Returns the number of rows of Mat

def Nrows: Int // Returns the number of rows of Mat

def Ncols: Int // Returns the number of columns of Mat

Accessing the internal implemenation:

The **Array[Array[Double]] v** keeps the Matrix internally and can be accessed as:

var ddArr = myMat.getv

Matrix number of Rows and can be retrieved as:

var Nrows = myMat.length

var Nrows = myMat.getv.length

var Ncols = myMat.getv(0).length

var siz = size(myMat) // returns an array siz such that:

// siz(0): #rows, siz(1): # columns

Resizing:

myMat(14,15)=1.111 // automatically resizing to include the element

Convenient Matrix MATLAB-like operations:

a. Row selection

- Rows from Matrix *M* can be selected as ***M(fromRow::toRow, ::)***, where *fromRow* is the starting row, *toRow* is the ending row, e.g.

  var mr2_5 = myMat(2::5, ::) // select rows 2 to 5, all columns, i.e. MATLAB's myMat(2:5,:)

- Order of row selection is **reversed** if *fromRow > toRow, e.g.*

  var mr5_2 = myMat(5::2, ::) // select rows 5 downto, all columns, i.e. MATLAB's myMat(5:-1:2.:)

- Step to skip rows can be specified as ***M(fromRow, incR, toRow, ::),*** where *fromRow* is the starting row, *incRow* is the step with which the selection of rows proceeds and *toRow* is the ending row, e.g.

var mr2_2_8 = myMat(2::2 :: 8, ::) // select starting from row 2, increment row index by 2 and ending at row 8, select all columns, i.e. MATLAB's myMat(2:2:8, :)

var mr9_2_3 = myMat(9::2 :: 3, ::) // i.e. MATLAB's myMat(9:-2:3,:)

- Columns from Matrix *M* can be selected as ***M(::, fromCol::toCol)***, where *fromCol* is the starting column, *toCol* is the ending column, e.g.

  var mc3_6 = myMat(::, 3::6) // select columns 3 to 6, all rows, i.e. myMat(:, 3:6)

- Order of column selection is **reversed** if *fromCol > toCol, e.g*.

  var mc6_3 = myMat(::, 6::3) // select columns 6 downto 3, all rows, i.e. myMat(:, 6:-1:3)

- Step to skip columns can be specified as ***M(::, fromCol::incC::toCol),*** where *fromCol* is the starting column, *incC* is the step with which the selection of columns proceeds and *toCol* is the ending column, e.g.

  var mc3_2_8 = myMat(::, 3:: 2 :: 9) // select starting from column 3, increment column index by 2 and ending at column 9, select all rows

- Selection of a compact rectangular range of the Matrix can be specified as ***M(fromRow, toRow, fromCol, toCol),*** e.g.

  var m2_5_4_7 =myMat(2::5, 4::7) // i.e. MATLAB's myMat(2:5, 4:7)

  var m5_2_4_7 =myMat(5::2, 4::7) // reversing rows, i.e. myMat(5:-1:2, 4:7)

  var m2_5_7_4 = myMat(2::5, 7::4) // reversing columns, i.e. myMat(2:5, 7:-1:4)

  var m5_2_7_4 = myMat(5::2, 7::4) // reversing both rows and columns, i.e. myMat(5:-1:2, 7:-1:4)

- Selection of a Matrix subrange specifying inrement at both rows and columns:

  var mR = myMat(2:: 2 ::7, 8 :: 2:: 4) // i.e. MATLAB's myMat(2:2:7, 8:-2:4)

Basic Matrix Routines

- **ones1(N: Int)** – returns a NXN Matrix filled with ones, **ones1(N: Int, M: Int)** – returns a NXM Matrix filled with ones
- **zeros1(N: Int)** – returns a NXN Matrix filled with zeros, **zeros1(N: Int, M: Int)** – returns a NXM Matrix filled with ones
- **diag1(N) –** returns an NXN diagonal matrix with ones at the diagonal
- **fill1(N: Int, val: Double)** – returns a NXN Matrix filled with **val**, **fill(N: Int, M: Int)** – returns a NXM Matrix filled with ones
- **sin(a: Matrix): Matrix, cos(a: Matrix): Matrix, tan(a: Matrix): Matrix, sinh(a: Matrix): Matrix, cosh(a: Matrix): Matrix, tanh(a: Matrix): Matrix, asin(a: Matrix): Matrix, acos(a: Matrix): Matrix, atan(a: Matrix): Matrix:** trigonometrical routines
- **abs(a: Matrix): Matrix, round(a: Matrix): Matrix, floor(a: Matrix): Matrix, ceil(a: Matrix): Matrix, sqrt(a: Matrix): Matrix, pow(a: Matrix, val: Double): Matrix, log(a: Matrix): Matrix, log2(a: Matrix): Matrix, log10(a: Matrix): Matrix, exp(a: Matrix): Matrix, toDegrees(a: Matrix): Matrix, toRadians(a: Matrix)**
- **dot(a: Matrix, b: Matrix): Matrix:** dot product of the two matrices, can be written also as: **a dot b**

Aggregation Routines

- **Columnwise Sum, sum(a: Matrix): Array[Double]**
  returns the columnwise sums as an Array[Double], e.g.

var a = ones1(4, 8)
var ac = sum(a)

- **Columnwise Mean, mean(a: Matrix): Array[Double]**
  returns the columnwise means as an Array[Double], e.g.
  var a = ones1(4, 8)
  var acm = mean(a)

- **Columnwise Product, prod(a: Matrix): Array[Double]**
  returns the columnwise products as an Array[Double]
- **Columnwise Mins, min(a: Matrix): Array[Double]**
  returns the columnwise minimums as an Array[Double]
- **Columnwise Maxs, max(a: Matrix): Array[Double]**
  returns the columnwise maximums as an Array[Double]

- **Rowwise Sum, sumR(a: Matrix): Array[Double]**
  returns the rowwise sums as an Array[Double]
- **Rowwise Mean, meanR(a: Matrix): Array[Double]**
  returns the rowwise means as an Array[Double], e.g.
  var a = ones1(4, 8)
  var acm = meanR(a)

- **Rowwise Product, prodR(a: Matrix): Array[Double]**
  returns the rowwise products as an Array[Double]
- **Rowwise Mins, minR(a: Matrix): Array[Double]**
  returns the rowwise minimums as an Array[Double]
- **Rowwise Maxs, maxR(a: Matrix): Array[Double]**
  returns the rowwise maximums as an Array[Double]

Routines

def ones1(n: Int): Matrix - constructs a nXn Matrix of ones

def ones1(n: Int, m: Int): Matrix - constructs a nXm Matrix of ones

def zeros1(n: Int): Matrix - constructs a nXn Matrix of zeros

def zeros1(n: Int, m: Int): Matrix - constructs a nXm Matrix of zeros

def rand1(n: Int): Matrix - constructs a nXn Matrix of random values

def rand1(n: Int, m: Int): Matrix - constructs a nXm Matrix of random values

def eye1(n: Int): Matrix - constructs a nXn Matrix of zero everywhere except the diagonal

d**ef eye1(n: Int, m: Int): Matrix** - constructs a nXm Matrix of zero everywhere except the diagonal

Transpose operator ~ transposes the Matrix, e.g. var mt = m~

Description of the ScalaSci.Mat class

The ScalaSci.**Mat** class implements the default **zero-indexed** two-dimensional dense matrices in ScalaSci. We note that there are also some other **Mat** classes that implement zero-indexed matrices on-top of specific

libraries, e.g. EJML, MTJ., Apache Common Maths etc.

*It is important to note that the zero-indexed Matrix type is the **only library dependent Matrix** in ScalaLab. All the zero-indexed Matrix classes, e.g. ScalaSci.EJML.Mat, ScalaSci.MTJ.Mat etc, offer similar interfaces, but of course the functionality is based on the underlying Java library that the Scala Mat class wraps.*

Constructors

Mat(rows: Integer, cols: Integer) // Creates a Mat of size rows, cols initialized to zeros

var m=new Mat(2, 3)

Construction by specifying the initial elements, e.g.

var m1 = M0("4.5 5.6 -4; 4.5 3 -3.4") // space separated

var m2 = M0("4.5, 5.6, -4; 4.5, 3, -3.4") // comma separated

// construct Mat by copying the array values

Mat(da: Array[Array[Double]]) // Creates a Mat initialized with the da array

var dd = Array.ofDim[Double](2,4)

dd(1)(1)=11

var mdd = new Mat(dd)

Mat(rows: Integer, cols: Integer, initValue: Double) // Creates a Mat of size rows, cols initialized to initValue

var m=new Mat(3, 4, 3.4)

// construct a Mat from a two-dimensional double array without copying the array values

Mat(da: Array[Array[Double]], flag: Boolean) // Creates a Mat initialized with a reference to the da array

// construct a Mat from an one-dimensional double array

Mat(da: Array[Double])

var od = new Array[Double](10)

od(2)=2.2

var odm = new Mat(od)

Basic Methods

def size: Int // Returns the number of rows and columns of the Mat

def length: Int // Returns the number of rows of Mat

def Nrows: Int // Returns the number of rows of Mat

def Ncols: Int // Returns the number of columns of Mat

Routines

def ones0(n: Int): Mat - constructs a nXn Mat of ones

def ones0(n: Int, m: Int): Mat - constructs a nXm Mat of ones

def zeros0(n: Int): Mat - constructs a nXn Mat of zeros

def zeros0(n: Int, m: Int): Mat - constructs a nXm Mat of zeros

def rand0(n: Int): Mat - constructs a nXn Mat of random values

def rand0(n: Int, m: Int): Mat - constructs a nXm Mat of random values

def eye0(n: Int): Mat - constructs a nXn Mat of zero everywhere except the diagonal

d**ef eye0(n: Int, m: Int): Mat** - constructs a nXm Mat of zero everywhere except the diagonal

Transpose operator ~ transposes the Matrix, e.g. *var mt = m~*

Chapter 3 ScalaLab Libraries

Interfacing Core Java Scientific Libraries

The content of this section is rather technical and can be skipped by a reader that intends only to use ScalaLab.

ScalaLab provides a novel feature: a uniform high-level interface to multiple basic Java numerical libraries that can be switched in order the user to investigate their relative benefits easily. Each such interface implements the functionality of the *ScalaSciMatrix* trait, therefore a common set of functionality is enforced.

A fundamental task for any mathematical programming environment is to provide powerful classes for handling matrices. ScalaLab offers both a 1-indexed Matrix class and a switchable 0-indexed one. Moreover, ScalaLab offers and the powerful *RichDouble2DArray* type, that wraps the standard Java *two-dimensional* arrays, and provides a lot of MATLAB-like functionality.

Specifically, we utilize the NUMAL library [11] 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 that use one-indexed arrays. Thus, this is another additional reason to keep a one-indexed Matrix class. For the zero-indexed *Mat* class we can switch the library on which this class is based: some options are the JAMA library, the Efficient Java Matrix Library (EJML), the Matrix Toolkit for Java (MTJ) and the Apache Common Maths Library. Also, the Java translation of the classic book “Numerical Recipes in C++” [12] is utilized in order to provide additional functionality to the *RichDouble2DArray* class.

Library independent uniformity is accomplished by implementing similar interfaces with the corresponding Scala wrappers to these libraries. Also, we take care to make the library routines MATLAB-like whenever possible, to facilitate users familiar with MATLAB. As noted, a common set of functionality is enforced by the compiler with the *ScalaSciMatrix* trait.

The architecture of interfacing Java scientific code to ScalaLab consists of the following basic steps (Fig . 2):

I. A ***Wrapper Scala Class (WSC)*** is created that provides high level operations by defining operator like methods, e.g. '+' for addition. For example, the *EJML.Mat* Scala class provides a more convenient user friendly and uniform interface to the functionality of the *SimpleMatrix* Java class of the EJML library. Thus the main benefits of this class are twofold:

1. Providing operator-like methods, e.g. '+', '*' etc.

2. Designing similar methods for all interfaced libraries with the enforced implementation of the methods of the ScalaSci Matrix interface by the Scala compiler. These methods preferably are similar to MATLAB, in order the user to have a familiar and uniform interface.

II. The ***Scala Object for Static Math Operations (SOSMO)*** aims to provide short overloaded versions of the basic routines for each new type that interfaces a Java library. For example, it allows to use *sin(B)* where *B* can be an object of our *Mat* Scala class. An explanation for the utility of the SOSMO objects follows.

Although we have a single one-indexed matrix class implementation we have many possible zero-indexed ones. Routines as *sin(B)*, can be resolved by the compiler based on the type of B, e.g. *EJML.Mat* or *MTJ.Mat* type.

However, by convention important routines as *rand1(n, m)*, *ones1(n, m)* etc. operate on the **single** one-indexed matrix class and those ending with 0, i.e. *rand0(n, m)*, *ones0(n, m)* are based on **one of the many** zero-indexed matrix libraries.

Therefore, it is not convenient to have multiple zero-indexed matrices imported at the same time since these routines should cope with a particular matrix type. The interpreter needs to switch between different choices for the zero-indexed matrix class. Therefore, the rationale behind these objects is to **facilitate the switching of the Scala interpreter to a different set of libraries**. Each such object implements a large set of coherent mathematical operations. The interpreter simply needs to import the corresponding SOSMOs in order to switch functionality. For example, the file *StaticMathsEJML.scala* implements a SOSMO Scala object that has zero indexed matrix functionality based on the EJML library, thus *rand0(n, m)* returns a *Wrapper Scala Class (WSC)* matrix object suited for the EJML library. Similarly, the file *StaticMathsMTJ.scala* does similar things targeted at the MTJ library.

Concluding, the top level mathematical matrix functions e.g. *rand(int n, int m)*, *ones(int n)* etc., should cope with the Matrix type representation appropriate to the currently utilized library. Also, a matrix object denoted e.g. *Matrix* can refer to different matrices depending on the library. The "switching" of libraries is performed by initializing a different Scala interpreter that imports the corresponding libraries. Currently, there exists a class *StaticMaths* that performs important initializations for the JAMA library, *StaticMathsEJML* for the Efficient Java Matrix Library, *StaticMathsMTJ* for the Matrix Toolkit for Java (MTJ) and *StaticMathsCommonMaths* for the Apache Common Maths library. Thus, the utilization of the JAMA library is accomplished by creating a fresh Scala Interpreter that imports the *StaticMathsJAMA* object while for the EJML the *StaticMathsEJML* is imported.

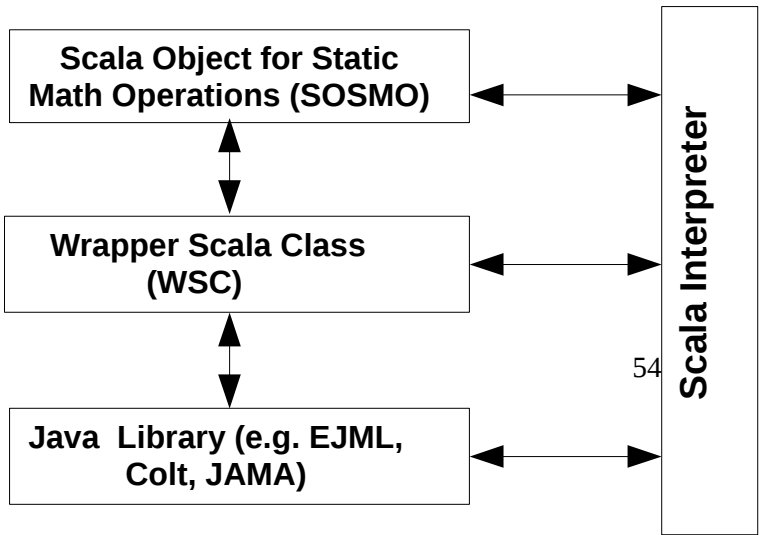


Fig 2. Interfacing libraries to ScalaLab

We investigated the integration of important Numerical Analysis libraries within the ScalaLab kernel. Although, the main executable file of ScalaLab (e.g. *ScalaLab.jar*) has some very important mathematical routines, most of them reside in separate JAR files that are placed in the lib folder. These additional libraries are automatically detected at startup and they are placed at the classpath of the Scala classloader.
However, for more specialized libraries of scientific code another mechanism is designed that utilizes them dynamically as toolboxes without embedding any code within the ScalaLab binary.

Ordering of imports can be significant

In *ScalaSci* the trait *ScalaSci.StaticScalaSciGlobal* defines some **global static operations** available in ScalaSci, **independent** of the library that the interpreter uses.
Other static operations exist for example in objects *RichDouble1DArray*, *RichDouble2DArray*, *Vec*, *Matrix*, *Sparse*, *CCMatrix* which are always imported.
The basic problem that this trait solves, is to import properly overloaded versions of a method. For example, the method *sin()*, has some fixed signatures independent of the particular library, which at the time of writing are (of course we can extend the list with other matrix types, for example sparse matrices):

sin(x: Double)
sin(x: Array[Double])
sin(x: Array[Array[Double]])
sin(x: Matrix)
sin(x: Vec)

All the *Scala Objects for Static Math Operations (SOSMOs)* should mixin this trait in order to have the common functionality. Additionally, each SOSMO should implement its library specific routines.
For example the *StaticMathsEJML* object, by mixing the *StaticScalaSciGlobal* trait, acquires all the implementations of the *sin()* for the library independent types.
Then, it implements the *sin()* for the EJML matrix as:

```
def sin(x: Mat) = {
  ScalaSci.EJML.Mat.sin(x)
}
```

IMPORTANT:

New import statements can overwrite symbols with the same name acquired from previous imports. Therefore, usually imports of the SOSMOs should be placed after objects containing symbols with the same names.

For example, if we write:

```
import ScalaSci.EJML.StaticMathsEJML._
import ScalaSci.Matrix._
```

then the statement:

```
var x = sin(3.4)
```

fails, while the statements:

```
var g = ScalaSci.EJML.StaticMathsEJML.rand0(2,3) // g is EJML matrix
sin(g)
```

succeed.

The problem is that the second import statement (i.e. import *ScalaSci.Matrix._*) imports the *sin()* from the *ScalaSci.Matrix* class that is defined only for *ScalaSci.Matrix*, and overwrites the *sin()* from the *ScalaSci.EJML.StaticMathsEJML*, that defines many more overloaded versions of *sin()* . So, the proper order of the imports is:

```
import ScalaSci.Matrix._
import ScalaSci.EJML.StaticMathsEJML._
```

Designing a high-level interface: a case study for the EJML library

This section elaborates 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. Also, the content is rather technical and can be skipped by the reader that wants only to learn to use ScalaLab as a tool.

The *ScalaSci.EJML.Mat* (abbreviated *Mat)* class in ScalaLab wraps the Efficient Java Matrix Library (EJML, http://ejml.org/wiki/index.php?title=Main_Page) *SimpleMatrix* class, allowing us to perform high-level MATLAB-like operations.

The *default (primary)* constructor performs global initialization operations (since in Scala any auxiliary constructor must call finally the primary constructor). It keeps a reference to the SimpleMatrix object that is the link to EJML's library functionality. This primary constructor is shown:

```
class Mat( smi: SimpleMatrix)  extends AnyRef with ScalaSci.ScalaSciMatrix[ScalaSci.EJML.Mat] = {
  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 }
```

Trait *ScalaSciMatrix* enforces common functionality to all the ScalaSci matrices, independently of the library that use.

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

The operator '+' in *c = a+b* is an *infix* operator. Scala also 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. Then for a variable *A* of Matrix type, the expression *A~* evaluates to the transpose of A.

Convenient infix operators are implemented in ScalaLab for all matrix types, include all the basic operators, e.g. addition, subtraction, multiplication etc. This allows the user to write the same MATLAB-like code independently of the utilized Java library, e.g. code such as:

```
var  a = rand0(4,5)   // a 4X5 random matrix
```

var b = rand0(5, 8) // a 5 X 8 random matrix

*var ab = a*b // perform matrix multiplication*

*var cc = 4.7*a-100 // an expression involving numbers and matrices*

var ma = -a // unary minus, note that the space before -a is required, to differentiate from =-

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 situation when a Mat is added to a double array. Therefore, the situation 4.8+A is more difficult than A+4.8, that can be implemented by defining a method '+' for the class of object A. 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 [15] 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.

ScalaLab defines automatically a large number of implicit conversions but also each application can define their own. 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 (this implicit conversion were in our original design that we modified):

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

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

var a = rand0(200, 300) // creates a 200 by 300 Matrix

var a2 = 2+a // performs the addition by implicitly converted 2

the 2 at the initial design was transformed to a 200X300 Matrix filled with 2, and the operation performed as Matrix addition.

However, this design was not very elegant and also not as effective that 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 ScalaSci, e.g. with *Mat, Matrix, EJML.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 has defined an operation to add a Matrix. Therefore, the operation proceeds effectively without allocating any new space (at the original design a Matrix object was created filled with 2s).

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

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 1s

var b = 10+a // add the value 10 to all the elements returning b as an Array[Array[Double]]

*var c = b + 89.7*a // 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 note that the *RichDouble2DArray* class is a powerful one providing a lot of MATLAB-like functionality to the Array[Array[Double]] type.

Scala programmers can implement 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 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, 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 = {

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, e.g. *M(2, k, m, 4, 2, N)* to extract a range denoted in MATLAB as *M(2:k:m, 4:2:N).*

For enhancing the efficiency of numerical code ScalaSci can offer two additional types of operators (currently these type of operators are experimental):

a. In-place operators.

These operators are mutable in that they act upon the receiver without creating and returning a copy of it, e. g.

var a = vones(3) // a 3-element vector of 1, i.e. a = [1.0,1.0, 1.0]

a ++ 4 // adds 4 to all the elements of the vector, i.e. a = [5.0, 5.0, 5.0]

*S*ince such in-place operators avoid the creation of new objects and the subsequent garbage collection overhead they have significant performance benefits, e.g. can be 3 to 5 times faster.

b. Right-associative operators.

Operations like e.g.

var aa = 10+a

are performed in ScalaLab with implicit conversion of the receiver (i.e. of 10 in this case) to a type for which the operation is valid. The implicit conversion however has some overhead. In order to avoid it the user can use right associative operators. Specifically, an operation that ends in ':' associates to the right, thus

var aaa = 10 +: a

is evaluated as a.+:(10)

and thus is a valid operation.

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*. 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. Therefore by creating global objects we have the same convenience as if global methods existed. For example, all the variety of plotting functions are available with their simple name (e.g. *plot(..), figure(..)*) by importing a relevant plot object.

Developing Code with ScalaSci Classes

ScalaSci classes are Scala language classes that can also exploit the additional support of ScalaLab for scientific code, i.e. the specialized constructs, operators and all the libraries.

We describe first the case of building simple **Scala** code and then the little more complicated case of developing **ScalaSci** code**,** i.e. Scala code that utilizes the expanded ScalaLab functionality.

a. Building simple Scala code with Scala Classes.

Suppose that you want to implement the Gaussian basis function.

Follow these steps:

1. Create a directory to place your code, e.g. "C:\neural"
2. Create a file named e.g. "myGaussClass.scala" .
3. Implement your class as for example:

   class gauss(val mi: Double, val sigma: Double)

   {

   def gs (x: Double) = {

   Math.exp(-(x-mi)(x-mi))/(2*sigma*sigma)*

   }

   }

4. Compile your "myGaussClass.scala" file either from the command line or from the *ScalaLab Explorer.* The later option is preferable since the Scala compiler integrated within the *ScalaLab* is used and thus even a Scala installation is not required. The compilation produces the class *gauss.class.*

5. In order your new class to be accessible, the ***classpath*** of the **Scala interpreter** (i.e. the **ScalaClassPath)** should be updated. You can update the ScalaClassPath in two ways:

a. Press the "*Update ScalaClassPath*" button or right-mouse click at the combo box placed directly below that button. A browsing file dialog appears where you can select the source file of your code (e.g. "gaussian.scala") in order to update the **ScalaClassPath.**

b. Alternatively, if you display the path at the **ScalaLab Explorer** you can right mouse click on it, and use the “Paths” popup menu option to update the **ScalaClassPath** with your path.

6. Given that the **ScalaClassPath** has been updated to include the “c:\neural” folder, and *Gaussian.class* is now accessible. You can use it for example as:

```
var mi=10.3; var sigma = 2.7

var tm = inc(2, 0.01, 30)  // time axes

var res = new Vec(length(tm))

var myGaussObj = new gauss(mi, sigma)

for (k<-0 to length(tm)-1) res(k) = myGaussObj.gs(tm(k))

plot(tm, res)
```

ScalaLab ClassPath / SourcePath

The **ScalaLab ClassPath** controls where the Scala compiler looks for *.class* files. Therefore, it is required to set it properly in order to be able to execute compiled user code.

The ScalaLab SourcePath allows the Scala interpreter to locate Scala scripts and to compile them.

The important variable that we must consider is the ScalaLab ClassPath since it controls the visible class files to the Scala interpreter.

Currently a single setting controls both the ScalaLab ClassPath and ScalaLab SourcePath.

To append your directories to the ScalaLab ClassPath do the following from the ScalaLab Explorer window:

Click on the button **“Update ScalaClassPath to include selected files**”. On the browsing file dialog that appears select any file within your directory that you want to append. The “ScalaSci Script/Classes Directories” combo box is updated to include the specified directory. Selecting any directory from the former combo box opens it in the ScalaLab Explorer.

You can also select the path with a right-mouse click from the file system view presented at the right-up JTree based view. Then the **Paths** menu has the corresponding option to update the path.

Toolboxes of scientific code

An important advantage of ScalaLab is its ability to cope easily with a MATLAB-like scripting with all the existing libraries of Java scientific code. Therefore, we can use and extend the large base of Java scientific libraries, with the powerful facilities of scripting, inspecting variable contents and plotting results.

ScalaLab exploits the *reflection* capabilities of the Java Virtual Machine [2] in order to interrogate dynamically the Java libraries and to present graphically their class contents. After installing a toolbox (using the "**ScalaSci Toolboxes"** tab) we can display its classes by clicking on it. However, the installation of a toolbox creates a fresh Scala interpreter with a Java class loader that has the toolbox at its classpath. Therefore, the user needs to pay some attention at the installation of toolboxes since the current work context can be lost. Perhaps, a convenient feature is that the toolboxes that are placed within the ***defaultToolboxes*** folder (before ScalaLab's startup) are available without any installation.

An installed toolbox during one ScalaLab session, remains on the **ScalaClassPath**, e.g. having installed weka.jar, weka.jar remains on the ScalaClassPath for next sessions. We can easily **remove** an installed toolbox, using the ScalaLab Explorer. The installed toolboxes are displayed at the nodes under the **"Global Path Variables".**

Toolboxes can actually be removed with two methods:

1. Selecting the toolbox from the JTree at the ScalaLab explorer that displays the **"Global Path Variables".** This is the easier and most straightforward method.
2. Right-mouse click and selecting the remove option of the popup menu.

Chapter 4: Example Applications of ScalaLab Toolboxes

We describe examples of ScalaLab mostly by using also ScalaLab toolboxes. We should note that the framework of ScalaLab toolboxes is powerful, since it allows the scientist to use more effectively and conveniently the excellent scentific libraries developed for the Java Virtual Machine (most of them in the Java language).

The FastICA toolbox

The FastICA algorithm exploits the notion of non-Gaussianity, which is a requirement of independent-component analysis [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 nonnegative, 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 **X** that is known to be non-Gaussian. The *negentropy* of **X** is formally defined by

$$N(X)=H(X_{Gaussian})-H(X)$$

where $H(X)$ is the differential entropy of **X,** and $H(X_{Gaussian})$ is the differential entropy of a Gaussian random vector whose covariance matrix is equal to that of **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 utilized. Hyvarinen and Oja have proposed the approximation

$$N(V)=E[\Phi(V)]-E[\Phi(V)]^2$$

In order to utilize the FastICA toolbox, first download the *fastICA.jar* file from the ScalaLab's Google site (i.e. https://sourceforge.net/projects/scalalab/).

Then use the “ScalaSci Toolboxes” tab. The first step is to specify the ScalaLab toolbox with the “Specify toolboxes” button. After that, we should create a Scala interpreter that will have the toolbox at its classpath. Currently, we support two Interpreter types, the default one (i.e. button “Import toolboxes creating new Interpreter”), and the EJML Interpreter (i.e. button “Import toolboxes creating new EJML Interpreter). If we want to change the interpreter type (e.g. Creating an MTJ Interpreter, we can do latter, the new Interpreter will contain the toolbox configuration.

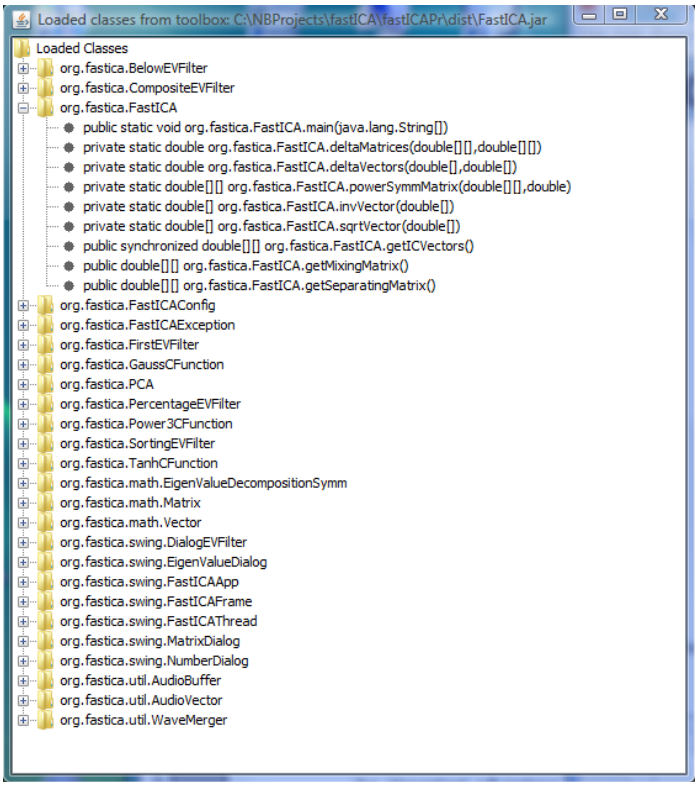
Finally, you can exploit the functionality of the toolbox in your applications by writing code such as:

Listing 1 : ScalaSci code that utilizes the FastICA toolbox

```
//          Creates a FastICA standard configuration:

var t = inc(0, 0.01, 10)
var N = t.length  // signal size
var sig1 = sin(2.3*t)+1.23*cos(0.663*t)
var sig2 = 6.7*sin(5.3*t)-1.23*cos(4.3*t)
var sig3 = 0.7*sin(0.53*t)+1.23*cos(0.43*t)
var sig4 = sin(5.2*t)
var noise = vrand(N)

var mixingMatrix = Array.ofDim[Double](5, 5)
mixingMatrix(0, 0) = 0.5;  mixingMatrix(0, 1) = 0.5;  mixingMatrix(0, 2) = 0.7;  mixingMatrix(0, 3) = 0.25;
mixingMatrix(0, 4) = 0.53;
mixingMatrix(1,0) = 0.3;  mixingMatrix(1, 1) = 0.7; mixingMatrix(1, 2) = 0.7;  mixingMatrix(1, 3) = 0.25;
mixingMatrix(1, 4) = 0.53;
 mixingMatrix(2, 0) = 0.6; mixingMatrix(2, 1) = 0.2;  mixingMatrix(2, 2) = 0.1;  mixingMatrix(2, 3) = 0.45;
mixingMatrix(2, 4) = 0.3;
mixingMatrix(3, 0) = 0.2; mixingMatrix(3,1) = 0.6;    mixingMatrix(3, 2) = 0.2;  mixingMatrix(3, 3) = 0.2;
mixingMatrix(3, 4) = 0.3;
 mixingMatrix(4, 0) = 0.3; mixingMatrix(4, 1) = 0.5; mixingMatrix(4, 2) = 0.3;  mixingMatrix(4, 3) = 0.54;
mixingMatrix(4, 4) = 0.9;

//  matrix with the signals before the mixing
 var initialSigs  = Array.ofDim[Double](5, 1)
 initialSigs(0) = sig1.getv;  initialSigs(1) = sig2.getv;  initialSigs(2) = sig3.getv
 initialSigs(3) = sig4.getv;  initialSigs(4) = noise.getv;

// perform mixing of signals
var mixedSignal = mixingMatrix * initialSigs

// join some filters into a standard filter
import org.fastica._
var filter = new CompositeEVFilter()
 filter.add(new BelowEVFilter(1.0e-8, false))
 filter.add(new SortingEVFilter(true, true))
   // build a ICA configuration
/*  Creates a FastICA configuration.
 * @param numICs the number of independent components
```

```
 * @param approach deflation or symmetric approach algorithm
 * @param stepSize the step size of an approach
 * @param epsilon accuracy exit condition
 * @param maxIterations the maximum number of iterations
 * @param initialMixingMatrix the initial mixing matrix, can be <code>null</code>
 */
 var numICs = 5
 var config = new FastICAConfig(numICs,  FastICAConfig.Approach.DEFLATION, 1.0, 1.0e-16, 1000, 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 fica = new FastICA(mixedSignal.getv, config, new Power3CFunction(), filter, listener)
    println();

var ICAVectors= fica.getICVectors
figure(1);  subplot(5,1,1); plot(sig1); subplot(5,1,2); plot(sig2);
subplot(5,1,3); plot(sig3);  subplot(5,1,4); plot(sig4);
subplot(5,1,5); plot(noise); title("Original Signals");

figure(2);   subplot(5,1,1); plot(ICAVectors(0)); subplot(5,1,2); plot(ICAVectors(1));
subplot(5,1,3); plot(ICAVectors(2));  subplot(5,1,4); plot(ICAVectors(3));
 subplot(5,1,5); plot(ICAVectors(4)); title("ICA Seperated");

figure(3); subplot(2,1,1);   plot(sig1.getv, Color.RED, "sig1");   plot(sig2.getv, Color.GREEN, "sig2");   plot(noise.getv,
Color.BLUE, "noise"); title("Original Signals");
subplot(2,1,2);         plot(ICAVectors(0), Color.GREEN,    "sig2");    plot(ICAVectors(1),    Color.BLUE,    "noise");
plot(ICAVectors(2), Color.RED, "sig1");   title("ICA Seperated");
```

Figure 3 : Illustrating the classes of the FastICA toolbox with a Jtree.



Numerical Solution of Differential Equations

Having presented the main points concerning the design of ScalaLab, we proceed with an example from the important topic of the numerical solution of Ordinary Differential Equations (ODEs). The concepts that we will present apply similarly to other application domains, e.g. nonlinear optimization. The particular example also demonstrates the utilization of Java classes from within ScalaSci, since the ODE solving machinery is from the NUMAL[11] Java library. Furthermore, we demonstrate that the specifications of the system for integration, can also be implemented in Scala without any performance penalty.

A supporting wizard can conceal the details of the ODE library from the user and permits the concentration at the task of specifying the differential equations. This wizard then builds and integrates to the system the appropriate Java or Scala code from a high-level specification of the involved systems of differential equations. It hides the details of the particular Java interfaces that need to be implemented for each supported ODE solution technique (similarly wizards can be developed for other tasks, e.g. nonlinear optimization).

The wizard is available from the ScalaLab's graphical user interface through a top-level menu option. Thus, even scientists that do not have Java experience can easily explore ODE solutions.

The process of the efficient implementation of ODE evaluation code in ScalaLab is described by means of an application for the implementation of the Lorenz system of chaotic equations.

The instance of the Lorenz attractor that is simulated is described with the equations :

$$\frac{dx_1}{dt} = 10 \cdot (x_2 - x_1)$$
$$\frac{dx_2}{dt} = -x_1 \cdot x_3 + 143 \cdot x_1 - x_2$$
$$\frac{dx_3}{dt} = x_1 \cdot x_2 - 2.67 \cdot x_3$$

As stated, ScalaLab exploits the NUMAL library of numerical code[11]. Standard Runge-Kutta integration is implemented in this library and is exposed through the *AP_rke_methods* interface.

The corresponding Java code for the Lorenz differential equations can be implemented easily with a Java class that implements the *AP_rke_methods* interface as follows (Listing 2):

Listing 2 : The Lorenz class that implements the AP_rke_methods interface in Java

```
import java.text.DecimalFormat;
import numal.*;

public class Lorenz extends Object   implements AP_rke_methods {

 // function to evaluate the derivatives of the system of n equations
 // with  state vector y[] at time t
  public void der(int n, double t, double y[])  {
    double xx,yy,zz;
// get previous state
    xx=y[1];  yy=y[2];  zz=y[3];
// update state
    y[1] = 10*(yy-xx);
    y[2] = -xx*zz+143*xx - yy;
    y[3] = xx*yy - 2.67*zz;
   }

}
```

The same class can also be coded in Scala as:

Listing 3: The Lorenz class that implements the AP_rke_methods interface implemented in Scala

```
import numal._

class LorenzScala extends Object with AP_rke_methods
 {
def der(n: Int, t: Double, y: Array[Double]): Unit =
{
      // get previous state
  var xx=y(1);  var yy=y(2);   var zz=y(3);
        // update state
   y(1) = 10*(yy-xx);
   y(2) = -xx*zz+143*xx - yy;
   y(3) = xx*yy - 2.66667*zz;

 }

def out(n: Int, t: Array[Double], te: Array[Double],
          y: Array[Double], data: Array[Double]):Unit = { }
}
```

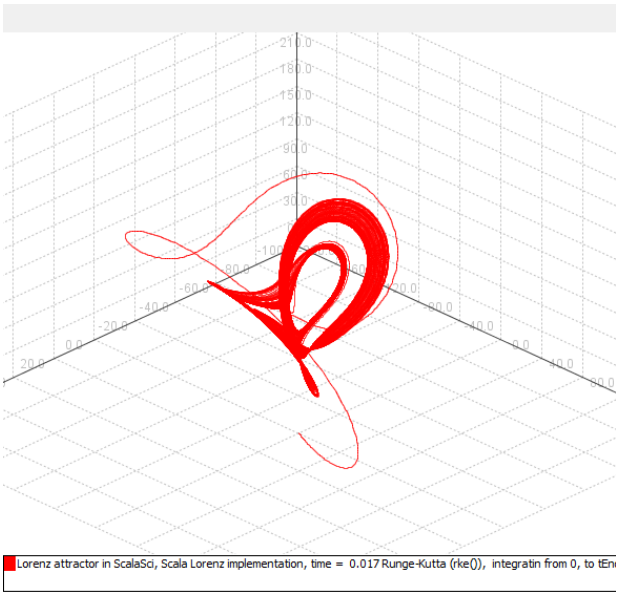


Figure 3 A plot obtained from the execution of the Lorenz example in ScalaLab

It is interested that the Scala implementation of the AP_rke_methods interface is slightly faster than the Java one! (about 10% faster)

The above class that implements the AP_rke_methods interface is easily produced automatically from the specification of the ODE equations by the ODE wizard graphical interface.
Clearly, it is an easy task to automate the construction of this class, and the ScalaLab's ODE Wizard provides an editable “template”. We can observe that the class implementation depends only upon the interfaces declared by the utilized Java library classes.
The automatic "injection" of pure Java class bytecode that implements the specific ODE system within a ScalaSci “driver” script requires:

1. Compilation of the classes (e.g. the *Lorenz* or *LorenzScala* class).
2. Placing the new class at a path accessible by ScalaLab*ClassPath*. The wizard facilitates the process by directly compiling and loading the automatically produced class (e.g. the class “Lorenz” at the example above). Also, the wizard can produce a “driver” ScalaSci script that integrates the system over a specified time domain (e.g. as that of Listing 4).

At the example ScalaSci code (Listing 4), the scripting environment requires the name of the Java or Scala class that implements the differential equations (at the example code at Line 1: *var lorenz2RKEObject = new LorenzScala*).
Subsequently, the function *rke()* that performs the numerical integration has a simple interface (line 2):

Analytic_problems.rke(x, xe, n, y, lorenz2RKEObject, data, fi, xOut, yOut)

It requires the starting value *x* of the independent variable of integration and the ending value, *xe*, until which the integration will be performed. The number *n* specifies the number of equations of the system, i.e. for the Lorenz system is 3. The vector **y** is the initial *state vector* of the system, initialized to random values in *(0, 1)*.
The function *rke()* is designed to return two 2-dimensional double vectors, i.e. *xOut* and *yOut*. The *xOut* vector is filled with the values taken by the independent variable ("*x*" in this case) and the *yOut* with the rest system state components.
The ScalaLab code for this integration task is listed below. Note, that in order to run this code we should compile the *LorenzScala* class first. Clearly, such scripts perplex knowledge of the numerical analysis library. However, they can be produced automatically from the ODE Class Wizard. The ODE Wizard already

supports some standard methods for ODE solution and we plan extend to support all the basic methods of the NUMAL library [11].

Listing 4: The ScalaSci script that integrates the Java code implementation of the Lorenz equations.

```
//   uses  the rke()  of the NUMAL library
//   the rke() routine solves an initial value problem for a system of first order ordinary differential
//   equations dy/dx = f(x,y), from x = x0 to x = xe where y(x0) = y0, by means of a 5-th
//   order Runge-Kutta method. The system is assumed to be nonstiff.

import scala._
import ScalaSci.Vec  // imports the Vector class
import ScalaSci.Vec._  // static members, e.g. ones(), zeros(), fill(), sin(), cos(), rand(),  etc.
import ScalaSci.Mat._     // static members, e.g. ones(), zeros(), fill(), sin(), cos(), rand(),  etc.
import java.util.Vector
import numal._     // numerical analysis library routines
import ScalaSci.math.plot.plot._     // plotting routines


var n = 3   // the number of equations of the system
var x = new Array[Double](1)    // entry:   x(0) is the initial value of the independent variable
var xe = new  Array[Double](1)   //  entry:  xe(0) is the final value of the independent variable
var  y = new  Array[Double](n+1)    // entry: the dependent variable, the initial values at x = x0
var data = new Array[Double](7)    // in array data one should give:
                              //     data(1):   the relative tolerance
                              //      data(2):  the absolute tolerance
                              // after each step data(3:6) contains:
                              //      data(3):  the steplength used for the last step
                              //      data(4):  the number of integration steps performed
                              //      data(5):  the number of integration steps rejected
                              //      data(6):  the number of integration steps skipped
                              // if upon completion of rke data(6) > 0, then results should be considered most criticallly

var fi = true;                    // if fi is true then the integration starts at x0 with a trial step xe-x0;
                              // if fi is false then the integration is continued with a step length data(3)* sign(xe-x0)


// a Java/Scala class that implements the AP_rke_methods interface should be specified
```

```
// The AP_rke_methods interface requires the implementation of two procedures:
//    void der(int n, double t, double v[])
//                this procedure performs an evaluation of the right-hand side of the syste4m with dependent variable v[1:n] and
//             and independent variable t; upon completion of der the right-hand side should be overwritten on v[1:n]
//    void out(int n, double x[], double xe[], double y[], double data[])
//             after each integration step performed, out can be used to obtain information from the solution process,
//             e.g., the values of x, y[1:n], and data[3:6]; out can also be used to update data, but x and xe remain unchanged



y(1) = 0.04840037112900297
y(2) = 0.13548442252861992
y(3) = 0.8792018004421086

     var xstart = 0   //  initial value of the independent variable
     var xend = 100 // final  value of the independent variable
     x(0) = xstart
     xe(0) = xend

     data(1) = 1.0e-6   // relative tolerance
     data(2) = 1.0e-6   // absolute tolerance

     var xOut:Vector[Array[Double]] = new Vector()
     var yOut:Vector[Array[Double]] = new Vector()

    // Line 1

  var lorenz2RKEObject = new LorenzScala

    tic()
    // Line 2
    Analytic_problems.rke(x, xe, n, y, lorenz2RKEObject, data, fi,  xOut, yOut)
    var timeCompute = toc()

   var plotTitle = "Lorenz attractor in ScalaSci, Scala Lorenz implementation, time =  "+timeCompute+ " Runge-Kutta (rke()),  integratin from "+xstart+", to tEnd= "+xend

  var color = Color.RED
```

figure3d(1); plotV(yOut, color, plotTitle)

A plot from the integration of the Lorenz equations is illustrated in Figure **3** .

Integration with the Apache Common Maths Library

We can also use the *Apache Commons Math* library that offers an elegant framework for numerical integration. The code has as follows:

Listing 5 The Lorenz system integrated with the Apache Commons Library

```
// these import statements concerning Apache Commons can be injected with the appropriate menu option
import org.apache.commons.math._
import org.apache.commons.math.analysis._
import org.apache.commons.math.analysis.function._
import org.apache.commons.math.analysis.integration._
import org.apache.commons.math.analysis.interpolation._
import org.apache.commons.math.analysis.polynomials._
import org.apache.commons.math.analysis.solvers._
import org.apache.commons.math.dfp._
import org.apache.commons.math.distribution._
import org.apache.commons.math.estimation._
import org.apache.commons.math.exception._
import org.apache.commons.math.exception.util._
import org.apache.commons.math.filter._
import org.apache.commons.math.fraction._
import org.apache.commons.math.genetics._
import org.apache.commons.math.geometry._
import org.apache.commons.math.geometry.euclidean.oned._
import org.apache.commons.math.geometry.euclidean.threed._
import org.apache.commons.math.geometry.euclidean.twod._
import org.apache.commons.math.geometry.partitioning._
import org.apache.commons.math.geometry.partitioning.utilities._
import org.apache.commons.math.linear._
import org.apache.commons.math.ode._
import org.apache.commons.math.ode.events._
import org.apache.commons.math.ode.nonstiff._
import org.apache.commons.math.ode.sampling._
import org.apache.commons.math.optimization._
import org.apache.commons.math.optimization.direct._
import org.apache.commons.math.optimization.fitting._
import org.apache.commons.math.optimization.general._
import org.apache.commons.math.optimization.linear._
import org.apache.commons.math.optimization.univariate._
import org.apache.commons.math.random._
import org.apache.commons.math.special._
import org.apache.commons.math.stat._
import org.apache.commons.math.stat.clustering._
import org.apache.commons.math.stat.correlation._
import org.apache.commons.math.stat.descriptive._
import org.apache.commons.math.stat.descriptive.moment._
import org.apache.commons.math.stat.descriptive.rank._
import org.apache.commons.math.stat.descriptive.summary._
import org.apache.commons.math.stat.inference._
import org.apache.commons.math.stat.ranking._
import org.apache.commons.math.stat.regression._
```

```
import org.apache.commons.math.transform._
import org.apache.commons.math.util._
import edu.jas.arith._
import edu.jas.poly._
import edu.jas.integrate._

class LorenzODE() extends AnyRef with FirstOrderDifferentialEquations {

var xx = 0.0; var yy = 0.0; var zz = 0.0
def getDimension = 3
def computeDerivatives(t: Double, y: Array[Double], yDot: Array[Double]) = {
 xx=y(0);  yy=y(1);  zz=y(2);
 yDot(0) = 10*(yy-xx);
 yDot(1) = -xx*zz+143*xx - yy;
 yDot(2) = xx*yy - 2.66667*zz;

}
}
var stepHandler = new StepHandler() {
def reset = { }


var cnt=0
var warmUpCnt=5
def handleStep( interpolator: StepInterpolator, isLast: Boolean)= {
var t = interpolator.getCurrentTime()
var y = interpolator.getInterpolatedState()
warmUpCnt -= 1
if (warmUpCnt < 0) {
 cnt += 1

vx = y(0) :: vx
vy = y(1) :: vy
vz = y(2) :: vz
}

}

def requiresDenseOutput = false
}


var vx = scala.List[Double]()
var vy = scala.List[Double]()
var vz = scala.List[Double]()
var dp853 = new DormandPrince853Integrator(1.0e-8, 100.0, 1.0e-10, 1.0e-10)
dp853.addStepHandler(stepHandler)
var ode = new LorenzODE()
var y = Array(0.0, 1.0, 1.0)
tic
dp853.integrate(ode, 0.0, y, 1600, y)
var tm=toc
plot(vx.toArray, vy.toArray, vz.toArray)
title("Integrated with Dormand Prince, time = "+tm+" num of points = "+vx.length)
```

Example on root finding

The *zeroin* method of the NUMAL library determines a zero of a function within an interval. For example, consider the problem of determining a zero of $e^{-3x}(x-1)+x^3$ within the interval[0, 1].

Listing 6 Using the *zeroin* method from NUMAL for root finding

```
// these NUMAL imports can be injected from the corresponding menu option
  import _root_.java.util.Vector ;
  import _root_.numal._ ;
  import _root_.numal.Algebraic_eval._;
  import _root_.numal.Analytic_eval._
  import _root_.numal.Analytic_problems._
  import _root_.numal.Approximation._
  import _root_.numal.Basic._;
  import _root_.numal.FFT._;
  import  _root_.numal.Linear_algebra._;
  import _root_.numal.Special_functions._;
  import java.text.DecimalFormat



class tzeroin extends AP_zeroin_methods {
  override def fx(x: Array[Double]) = Math.exp(-x(0)*3.0)*(x(0)-1)+x(0)*x(0)*x(0)
  override def tolx(x: Array[Double]) = Math.abs(x(0))*1.0e-6+1.0e-6
}

  var a = new Array[Double](1);
  var  b = new Array[Double](1);
  var zeroinClass = new  tzeroin();
a(0) = 0.0;
b(0) = 1.0;
var m = Analytic_problems.zeroin( a, b, zeroinClass);
System.out.println("Zero is  "+a(0)); // the wrapper class declaration follows

// verify the root graphically
var t = linspace(-1, 2, 200)
```

```
var x = exp(-t*3.0)*(t-1)+t*t*t
plot(t,x)
```

Exploiting Lower Level Functionality

ScalaSci implements many high-level Scala classes, that wrap usually Java classes of many scientific libraries. However, although it is more convenient to work with those Scala classes, some lower-level functionality is lost. We present 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 of programming 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. The key to developing this type of code is the *getLibraryMatrixRef* method of the *ScalaSciMatrix* class. The *getLibraryMatrixRef* method returns the library dependent class that implements native operations. This allows ScalaLab code to combine Scala implemented operations, 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 *getLibraryMatrixRef* is:

```
def getLibraryMatrixRef: AnyRef
```

This abstract method is implemented for the various concrete ScalaSci classes as:

```
def getLibraryMatrixRef() =  sm  // the ScalaSci.EJML.EJMLMat wraps an EJML SimpleMatrix
def getLibraryMatrixRef() =  dm // the ScalaSci.JBLAS.Mat class wraps the org.jblas.DoubleMatrix class, thus return simply the data representation
def getLibraryMatrixRef() = dm // the ScalaSci.MTJ.Mat class wraps the no.uib.cipr.matrix.DenseMatrix class, thus return simply the data representation
def getLibraryMatrixRef() = rm // the ScalaSci.CommonMaths.Mat class wraps the org.apache.commons.math.linear.Array2DRowRealMatrix class

 def getLibraryMatrixRef() = v // the ScalaSci.RichDouble2DArray does not wrap a Matrix class of a specific library, thus return simply the data representation
  def getLibraryMatrixRef() = v // the ScalaSci.Mat does not wrap a Matrix class of a specific library, thus return simply the data representation
 def getLibraryMatrixRef() = v // the ScalaSci.Matrix does not wrap a Matrix class of a specific library, thus return simply the data representation
```

Another important routine, the *matFromLibrary*, converts from the lower level matrix representation back to the higher level ScalaSci matrix. Therefore, the *getLibraryMatrixRef* 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 matFromLibrary

The matFromLibrary can be called using the ScalaSci matrix reference on which *getLibraryMatrixRef* is called, e.g.

```
var x = new ScalaSci.Mat(4,5)
var xv = x.getLibraryMatrixRef  // take the internal representation
xv(0)(0) = 200 // change the internal representation
var xrecons = x.matFromLibrary  // return an updated Matrix
```

Alternatively, we can perform computations in different internal matrix structures and then reconstruct, as e.g.

```
var x = new ScalaSci.Mat(4, 5)
var xv = x.getLibraryMatrixRef // take the internal representation
xv(0)(0) = 200 // change the internal representation
var xnew = Array.ofDim[Double](7, 8) // differently sized matrix
xnew(2)(3) = xv(0)(0)
var xrecons2 = x.matFromLibrary(xnew) // reconstruct a differently sized matrix
```

Also, very useful routines, for gluing multiple libraries are the following:

```
// converts any matrix type to a simple Array[Array[Double]]
def toDoubleArray()
// converts from an Array[Array[Double]] to any matrix representation
def fromDoubleArray(x: Array[Array[Double]])
```

For example:

```
var x = Array(Array(9.3, -0.3, 2.8), Array(4.5, 3.4, 2))
// construct an  EJML matrix
var xEJML = ScalaSci.EJML.StaticMathsEJML.fromDoubleArray(x)
// construct an MTJ matrix
var xMTJ = ScalaSci.MTJ.StaticMathsMTJ.fromDoubleArray(x)
// the constructed matrix type depends on which library we are switched on
var xy = fromDoubleArray(x)
// now convert back from the EJML matrix to double array
var xEJMLrecover = xEJML.toDoubleArray
// now convert back from the MTJ matrix to double array
var xMTJrecover = xMTJ.toDoubleArray
```

Below we provide an additional example of mixed type code for the EJML library.

We demonstate low-level EJML functionality by importing the static definitions of the *org.ejml.simple.SimpleMatrix* class and exploiting the native *SimpleMatrix*, which the *ScalaSci.EJML.Mat* wraps. The following code can be executed in ScalaLab directly, **however an EJML type of Interpreter should be active**.

```
import org.ejml.simple.SimpleMatrix._  // static methods of  EJML SimpleMatrix class
val x = rand0(8, 8)  // create an EJML.Mat random matrix
val xsm = x.getLibraryMatrixRef  // get a reference to the EJML SimpleMatrix, which the ScalaSci.EJML.Mat wraps
// demonstrating directly using the SimpleMatrix
val sm20 = identity(20)  // create a SimpleMatrix with 1s at the diagonal
val diagArray = Array(2.3, 7.8, 6.7)
val smDiagonal = diag(diagArray:_*)  // create a diagonal SimpleMatrix
// creates a new SimpleMatrix with random elements drawn from a uniform distribution from minValue to maxValue
val smr = random(4, 5, -4.0, 5.8, new Random())
// compute the invert matrix
var xsmi = xsm.invert
// wrap the invert SimpleMatrix to a ScalaSci.EJML.Mat
var xsmiToEJMLMat = new ScalaSci.EJML.Mat(xsmi)
// .. now we can verify more easily that the inversion succeded
var shouldBeIdentity = x*xsmiToEJMLMat
// and we can more conveniently construct the EJML matrix as
```

```
var reconsEJML = x.matFromLibrary(xsmi)
// if we perform changes directly on the native matrix reference we can reconstruct more conveniently,
// by calling directly the matFromLibrary() e.g.
xsm.set(0, 40); xsm.set(1, 3.3); // some direct changes to the low level matrix representation

var reconsDirectEJML = x.matFromLibrary
```

Fast Fourier Transforms: Benchmarking different FFT libraries

ScalaLab can use effective implementations of FFT from Java libraries. The following code tests: 1. an Oregon DSP based implementation, 2. the Apache Common Maths, 3. the Numerical Recipes implementation.

At the following listing the Apache Common Maths FFT and the Numerical Recipes FFT is performed only for real data. Also, an important observation is that by using float arrays instead of doubles we have not obtained any speedup, therefore in today's JVMs the more accurate double type does not seem to impose any overhead. Thus, we kept only the version of our routines for double arrays.

Here is the code to test the FFTs:

```
closeAll
var  N  = 16384
 var log2N = 14

var  xr =  new Array[Double](N)  // real part
 var  xi =  new Array[Double](N)  // imaginary part
 var  Xr =  new Array[Double](N)
 var  Xi =  new Array[Double](N)

var k=0
while (k< N)  {      xr(k) = rand;      xi(k) = rand; k+=1    }

 var Xfm = new DSP.fft.CDFT( log2N )
 var Xfmr = new DSP.fft.RDFT(log2N)

   // evaluate transform of data

var Nt = 100 // how many times to evaluate the FFTs
 tic
 k=0
 while (k < Nt) {
   Xfm.evaluate( xr, xi, Xr, Xi )
   k += 1
```

```
  }

   var tmOregon = toc

// FFT with real only data
    tic
   k=0
   while  (k < Nt) {
    Xfmr.evaluate( xr, Xr  )
     k += 1
   }
   var tmOregonR = toc

   figure(1); plot(Xr); title("Oregon DSP based FFT took "+tmOregonR)   // plot the resulting Fourier
transform



// test the Apache Commons FFT
   var fftXr2 = ScalaSci.FFT.ApacheFFT.fft(xr )
   tic
   k=0
   while (k < Nt)  {
    fftXr2 = ScalaSci.FFT.ApacheFFT.fft(xr )
    k += 1
}
    var tmApache = toc



// test the Numerical Recipes FFT
 var (nrR, nrIm) = ScalaSci.FFT.FFTScala.fft(xr)
   tic
  k=0
  while (k < Nt) {
    var (nrR, nrIm) = ScalaSci.FFT.FFTScala.fft(xr)
    k += 1
   }
  var tmNR = toc
```

```
figure(3); plot(nrR)
title("Numerical Recipes based FFT took "+tmNR)
```

The following script also compares the Numerical Recipes, JTransforms, Apache Common Maths and Oregon DSP FFT implementations. The results indicate that Oregon DSP seems the fastest, while Numerical Recipes FFT performs also very well.

```
// this script applies and compares different FFT implementations
// construct signal
closeAll   // close all figures
var log2N = 17
var N = 2^^log2N
var t = linspace(0, 1, N)  // sample interval [0,1] taking N points
var dx = t(1)-t(0)  // distance between successive points
var SF = 1/dx   // sampling frequency
var NF = 0.5*SF  // Nyquist Frequency

 var F1  = NF/20

 var x1= sin(F1*t)+0.66*cos(2*F1*t)

// Numerical Recipes FFT

tic
 var (rx, ix, freqs)  = ScalaSci.FFT.FFTScala.fft(x1, SF)

 var recons = ScalaSci.FFT.FFTScala.ifft(rx, ix)
var tmNR = toc
 figure(1);  subplot(3,1,1); plot(freqs, rx, "fft")
   subplot(3,1, 2); plot(x1, "original");
   subplot(3, 1, 3); plot(recons, "reconstructed")



   // JTransforms FFT
val dfft = new edu.emory.mathcs.jtransforms.fft.DoubleFFT_1D(N )

val data = new Array[Double](2*N)
```

```
val datafft = new Array[Double](2*N)
val cpdata = new Array[Double](2*N)
val reconsJT = new Array[Double](2*N)

// copy the signal
var k=0
while  (k < N)  {  data(k) = x1(k); k += 1 }

  // another copy
k=0;  while  (k < 2*N) {  cpdata(k) = data(k);  k += 1 }

  // perform a real FFT
tic
dfft.realForwardFull(data)
k=0;   while ( k < 2*N ) {   datafft(k) = data(k); k += 1 }

// perform an inverse FFT
dfft.complexInverse(data, true)

var rsig = new Array[Double](N)
 k=0;  while   (k < N)  { rsig(k) = data(2*k); k+= 1 }
 var tmJTransforms = toc

    figure(2); subplot(3,1,1); plot(x1, "original JTransforms");
  subplot(3, 1, 2); plot(rsig, "reconstructed");
  subplot(3, 1, 3); plot(datafft, "FFT")

  // Apache Commons FFT
import ScalaSci.FFT.ApacheFFT
import ScalaSci.FFT.ApacheFFT._

def getReal(x: org.apache.commons.math3.complex.Complex) = {x.getReal}
tic
var y = fft(x1)
var arecons = ifft(y)
var tmApache = toc
 figure(3); subplot(3,1,1); plot(x1, "original Apache Commons");
```

```
 subplot(3, 1, 2); plot(arecons map getReal, "reconstructed");

 subplot(3,1, 3); plot(y map getReal)



// Oregon DSP FFT
 var  Xfm = new DSP.fft.RDFT( log2N )
 var  osig = new Array[Double](N)
 var  ores = new Array[Double](N)
 tic
 Xfm.evaluate(x1, osig)
 Xfm.evaluateInverse(osig, ores)
 var tmODSP = toc
 figure(4);  subplot(3,1,1);   plot(x1, "Oregon DSP");
             subplot(3,1, 2);  plot(osig, "FFT");
             subplot(3,1, 3);  plot(ores, "Reconstructed");



             println("time NR = "+tmNR +", time JTransforms = "+tmJTransforms + ",  timeApache =
"+tmApache+", time Oregon DSP = "+tmODSP)
```

RBF Networks with the WEKA toolbox

This script demonstrates the utilization of the WEKA toolbox from within ScalaLab. It constructs an RBF network for the data set that is specified with an .arff file. That toolbox can be downloaded from the sourceforge ScalaLab site |(https://sourceforge.net/projects/scalalab/).

```
// this script  requires  WEKA toolbox to be first installed
import weka.classifiers.functions._
import weka.classifiers.Classifier
import weka.core._
import java.io._
import weka.core.converters.ConverterUtils._


var RBFNet = new RBFNetwork
var NClusters = 8
// set now RBF parameters
RBFNet.setNumClusters(NClusters)

var dataFile = getFile("Please specify your data with numeric data for the RBFNetwork")   // get
the datafile from the user

//var dataFile = "/home/sp/data/UCI/sonar.arff"
   //  read the datafile
var allData = DataSource.read(dataFile)
allData.setClassIndex(allData.numAttributes()-1)




// construct the training set and testing sets
var trainSet = new Instances(allData, 0)  // create an initial empty training set
var testSet = new Instances(allData, 0)    // create an initial empty testing set

var UseInTrain = true  // controls whether to add the Instance at the training set or at the testing
set
var enumInstances = allData.enumerateInstances()
while  (enumInstances.hasMoreElements) {
 var currInstance  = enumInstances.nextElement.asInstanceOf[Instance]
 if (UseInTrain)
    trainSet.add(currInstance)
 else
   testSet.add(currInstance)
 UseInTrain = !UseInTrain
}

RBFNet.buildClassifier(trainSet)   // build an RBF classifier on the training set
// test the classifier on the testing set
```

```
// extract the class labels
var enumTestInstances = testSet.enumerateInstances()
var numTestingInstances = testSet.numInstances()
var classLabels = new Array[Double](numTestingInstances)
var predictedLabels  = new Array[Double](numTestingInstances)
var predictedLabels2  = new Array[Double](numTestingInstances)


var cnt=0
var classIdx = testSet.classIndex   // get class index
while (enumTestInstances.hasMoreElements) {   // for all the elements of the testing set
  var currInstance = enumTestInstances.nextElement.asInstanceOf[Instance]
  var distForInstance = RBFNet.distributionForInstance(currInstance)
  var classOfInstance =  currInstance.toDoubleArray.apply(classIdx)
  classLabels(cnt) = classOfInstance
  predictedLabels(cnt) = distForInstance(0)
  predictedLabels2(cnt) = distForInstance(1)
  cnt += 1
}

figure(1)
linePlotsOn
hold("on")
plot(predictedLabels, Color.RED, "predicted");
plot(classLabels, Color.BLUE, "actual class"); title("rbf Network prediction")
```

Examples from discrete dynamical systems

The Logistic one-dimensional chaotic map

This section demonstrates the evaluation of simple discrete dynamical systems with ScalaLab. As a first example we consider the Logistic Map, $x_{k+1} = \lambda \cdot x_k \cdot (1 - x_k)$,λ=3.99, k=0,1,..

We demonstrate how we can evaluate these difference equations both as scripts (which is the simplest alternative), as compiled objects pasted directly over the workspace, and as compiled classes at the filesystem.

Implementation as pure script code

The code below demonstrates the evaluation of the Logistic map as a script.

```
close("all")  // closes all figures
val  L = 3.99    // the parameter of the chaotic logistic map
```

```
var N= getInt("Number of points for Logistic")  // get number of points to evaluate  from user

tic   // reset timer
var x = new Array[Double]( N)    // create an array to keep the results
x(0)=0.12   // initial state
  // evaluate the map equation
k=1
while (k<N) {
   x(k) =  L*x(k-1)*(1-x(k-1))
   k+=1
}

var tm = toc   // take timer contents, i.e. the elapsed time
plot(x)   // plot the computed values
title("Logistic Chaotic map evaluated for "+N+ " points =  +at time  "+tm)
```

Implementation as compiled objects

Since ScalaLab is based on the Java Virtual Machine (JVM) and the JVM uses the class files as program modules, we can develop the evaluation of the Logistic chaotic map application using a Scala object that evaluates the map. This approach is somewhat more difficult than the script approach, but has as benefits modularity, extensibility and reusability of the design. We should note that Scala objects are compiled to Java classes that have only static members. Scala objects can provide both data members and methods to the ScalaLab environment.

```
object Logistic {

  def  comp(L: Double, N: Int): Array[Double] = {     // declaration of the return type
Array[Double] is optional
    var x = new Array[Double]( N)   // allocate array to compute map's values
    x(0)=0.12   // initial condition
    var k=1
    while  (k < N) {
     x(k) =  L*x(k-1)*(1-x(k-1))
     k += 1
     }
```

```
  x  // x is the return  value from the method
 }
}
```

The *Logistic* object provides the method *comp(L: Double, N: Int)* that computes *N* points of the chaotic map $x_{k+1}=\lambda \cdot x_k \cdot (1-x_k)$ with parameter *L*. We explicitly declare the return type of the method as Array[Double]. We have also the option of not declaring the return type and to leave Scala's type inference mechanism to conclude it.

We can cut and paste the object above to the ScalaLab interpreter. The interpreter responds by displaying a message “defined module Logistic”. Therefore, our object has been compiled successfully and we can exploit it in our further computations. The script that follows uses this object to evaluate the map.

```
tic   // reset timer
var N=2000
var L=3.99
var x = Logistic.comp(L, N)     // use the compiled Logistic object to perform the computation
var tm = toc  // time required to perform the computation
plot(x)
title("Scala Time "+tm)
```

We can also import the Logistic object to the Interpreter in order to call the *comp* method directly. This version of the code has as follows:

```
import Logistic._
tic   // reset timer
var N=2000
var L=3.99
var x = comp(L, N)     // use the compiled Logistic object to perform the computation
var tm = toc  // time required to perform the computation
plot(x)
title("Scala Time "+tm)
```

We should note at this point that it is not necessary to compile these objects, anytime we have to use them. We can put their code in files, compile those files one time, and update the *ScalaLabClassPath* variable to include the directories of the compiled classes.

The Henon and Ikeda two-dimensional chaotic Maps

Similarly, we can easily develop ScalaLab code to evaluate the Henon and Ikeda two-dimensional discrete chaotic maps.

The code for Henon map is shown below:

```
close("all")  // or closeAll
val  A_CHAOS=1.4
val  B_CHAOS=0.3
var N=getInt("Number of points for Henon")
var x = Array.ofDim[Double](2, N)
x(0)(0)=0.12; x(1)(0) = 0.2;

tic
var k=1
while (k < N) {
  x(0)(k) = 1.0-A_CHAOS*x(0)(k-1)*x(0)(k-1)+x(1)(k-1)
  x(1)(k) = B_CHAOS*x(0)(k-1)
  k += 1
  }

var tm = toc
scatterPlotsOn  // display points only, not the connecting lines
plot(x)
title("Scala Time "+tm)
```

Also, the code for the Ikeda map follows:

```
// Ikeda map
close("all")
val R = 1; val C1 = 0.4; val C2 = 0.9; val C3 = 6
var N=200000
var x = new Array[Double]( N)
```

```
var y = new Array[Double]( N)
x(0)=0.12; y(0) = 0.2;

tic
var k = 1
var km = 0
var tau=0.0; var sintau=0.0; var costau=0.0
while (k< N) {
  km=k-1
  tau = C1-C3/(1+x(km)*x(km)+y(km)*y(km))
  sintau = sin(tau); costau = cos(tau);
  x(k) = R+C2*(x(km)*costau-y(km)*sintau)
  y(k) = C2*(x(km)*sintau+y(km)*costau)
 k += 1

 }

var tm = toc()
scatterPlotsOn();  // display points only, not the connecting lines
plot(x, y);
title("Scala Time for Ikeda map "+tm);
```

Bifurcation Diagram of the Logistic Chaotic Map

The following code plots a bifurcation diagram of the simple logistic chaotic map using direct plotting on Swing frame.

```
import _root_.java.awt._
import _root_.java.awt.event._
import _root_.javax.swing._
import _root_.javax.swing.event._
```

```
class bifLogistFrame extends JFrame  {
      setTitle("Bifurcation Diagram")
      var  xmax = 600
      var  ymax = 400
      setSize(xmax, ymax)

   override def  paint ( g: Graphics):Unit = {
      var xplot=0.0; var yplot=0.0

     xmax = size.width;  ymax = size.height
     g.clearRect(0, 0, xmax, ymax)
     g.setColor(Color.GREEN)
     var rstart = 2.0
     var r =  rstart  // starting value of the bifurcation parameter
      while (r <= 4.0) {
        xplot = xmax*(r-rstart)/rstart
        var x = 0.8
        var k=0  // warmup model for that bifurcation parameter
        while (k < 400) {   x = r*x*(1.0-x); k+=1 }

        k=0;
        while ( k < 400) {
          x = r*x*(1.0-x)
          yplot = ymax*(1.0-x)
          var m =  Math.round(xplot).asInstanceOf[Int]
          var n =  Math.round(yplot).asInstanceOf[Int]
          g.drawLine(m, n, m, n)
          k += 1
        }
        r = r+0.0005
      }
   }
}

 var f = new bifLogistFrame()
```

```
f.setVisible(true)
```

Examples of integrating ODEs

This section presents examples of using ScalaSci in various applications.

Listing 7 The Lorenz chaotic attractor evaluated with the multistep method from the NUMAL library

```
//   this ScalaSci file demonstrates the use of the multistep methods of the NUMAL library for integrating the Lorenz system

import java.util.Vector
import numal._     // numerical analysis library routines

// Java's Swing imports
   import _root_.java.awt._ ;
   import _root_.java.awt.event._ ;
   import _root_.javax.swing._ ;
   import _root_.javax.swing.event._;

   var n = 3;   // the order of the system

    var first = new Array[Boolean](1);   // if first is true then the procedure starts the integration with a first order Adams method
                                         //  and a steplength equal to hmin,  upon completion of a call, first is set to false
   first(0)=true

   var btmp = new Array[Boolean](2)
   var itmp = new Array[Int](3)
```

```
   var xtmp = new Array[Double](7)
   var x = new Array[Double](1)
   var y = new Array[Double](6*n+1)
   var ymax = new Array[Double](4)

    var save = new Array[Double](6*n+39)  //    in this array the procedure stores information which can be used in a continuing call
      // with first = false; also the following messages are delivered:
      //     save[38] == 0;  an Adams method has been used
      //     save[38] == 1;  the procedure switched to Gear's method
      //     save[37] == 0;  no error message
        //      save[37] == 1; with the hmin specified the procedure cannot handle the nonlinearity (decrease hmin!)
      //     save[36] ;  number of times that the requested local error bound was exceeded
         //       save[35] ;  if save[36] is nonzero then save[35] gives an estimate of the maximal local error bound, otherwise save[35]=0

   var jac = new Array[Array[Double]](n+1)
   var k=0
   while (k<=n) {
     jac(k) = new Array[Double](n+1)
     k += 1
}

    var xOut:Vector[Array[Double]] = new Vector()
    var yOut:Vector[Array[Double]] = new Vector()

   var hmin=1.0e-10
   var eps=1.0e-9

   y(1)=0.12; y(2)=0.3; y(3)=0.12;
   ymax(1) = 0.00001
   ymax(2) = 0.00001
   ymax(3) = 0.00001;

   var tstart = 0.0
   x(0) = tstart
 var xendDefault ="100.0"  // end point of integration, default value
```

```
var prompt = "Specify the end integration value"
var inVal  = JOptionPane.showInputDialog(prompt, xendDefault)
var tend = inVal.toDouble

var  javaClassName = "scalaExec.Functions.Chaotic.LorenzMultiStep"

var  invocationObject = Class.forName(javaClassName).newInstance();
var multistepObject = invocationObject.asInstanceOf[AP_multistep_methods]

tic()
    Analytic_problems.multistep(x, tend,y,hmin,5,ymax,eps,first, save, multistepObject, jac,
true,n,btmp,itmp,xtmp, xOut, yOut)

var  runTime =  toc()


var plotTitle = "Lorenz system, method Multistep,  ntegratin from "+tstart+", to tEnd= "+tend+", runTime =
"+runTime;

var color = Color.RED
figure3d(1); plotV(yOut, color, plotTitle)
```

Listing 8 The Double Scroll Attractor evaluated with the NUMAL library

```
import numal._    // numerical analysis library routines
import java.util.Vector

val tol =  0.0000000000000001
var  aeta = tol;  var reta = tol;

val n = 3;   // the number of equations of the system
var x = new Array[Double](1)   // entry:   x(0) is the initial value of the independent variable
var y = new Array[Double](n+1)   // entry: the dependent variable, the initial values at x = x0
aeta = tol  // aeta: required absolute precision in the integration process
reta = tol // reta: required relative precision in the integration process
var s = new Array[Double](n+1)
```

```
var h0=0.000001  // h0: the initial step to be taken

 var xOut:Vector[Array[Double]] = new Vector()
 var yOut:Vector[Array[Double]] = new Vector()

y(1)=0.4; y(2)= -0.3; y(3)=0.9;

x(0) = 0
var xe = 720
var javaClassName = "scalaExec.Functions.Chaotic.DoubleScrollDiffSys"
var  invocationObject = Class.forName(javaClassName).newInstance();

var diffSysObject = invocationObject.asInstanceOf[AP_diffsys_methods]

tic()
Analytic_problems.diffsys(x, xe, n, y, diffSysObject, aeta, reta , s, h0, xOut, yOut)
var timeCompute = toc()

var plotTitle = "Double Scroll attractor with ScalaSci, time "+timeCompute+ " end point = "+xe

var color = Color.RED

  figure3d(1); plotV(yOut, color, plotTitle);
```

Chapter 5 EJML Library

Introduction

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). ScalaLab implements an even higher-level and easier to use interface, the *ScalaSci.EJML.Mat* interface, while the expert user can still benefit from all the lower level ones.

These goals are accomplished by dynamically selecting the best algorithms to use at runtime and by designing a clean API. EML is free, written all in Java, and can be obtained from http://ejml.org/wiki/index.php?title=Main_Page

The EMJL 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 overwriting 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 seeked while for overdetermined systems (i.e. N > M) the least squares solution is computed.

The EJML library provides an extensive set of functionality implemented with the efficiency goal in mind. In ScalaLab we can utilize the basic algorithms even more easily with high-level mathematical notation and with a MATLAB-like interface.

ScalaLab utilizes by default for zero-indexed matrices the EJML (http://code.google.com/p/efficient-java-matrix-library/) library upon which builds a MATLAB-like easy to use way of performing matrix operations. It is important that the lower level routines of EJML are all accessible from ScalaLab, and we can call them, either when ScalaLab has not a higher-level equivalent or when we require maximum execution speed. Here we describe some aspects of working with the EJML library presenting many examples.

A high-level interface to the EJML library

The **EJMLMat** class implements **zero-indexed** two-dimensional dense matrices in ScalaSci that are based upon the Efficient Java Matrix Library.

Constructors

Mat(rows: Integer, cols: Integer) // Creates an EJML Mat of size rows, cols initialized to zeros
var m=new Mat(2, 3)
m.print // print the contents

Construction by specifying the initial elements, e.g.

var m1 = M("4.5 5.6 -4; 4.5 3 -3.4") // space separated
var m2 = M("4.5, 5.6, -4; 4.5, 3, -3.4") // comma separated

// construct Mat by copying the array values
Mat(da: Array[Array[Double]]) // Creates an EJML Mat initialized with the da array
var dd = Array.ofDim[Double](2,4)
dd(1)(1)=11
var mdd = new Mat(dd)
mdd.print // print finally the EJML Matrix

// construct a Mat by **reference to an existing double array**,
// in order to avoid duplicating the array
var tm = new Mat(1,1) // a small Mat object
tm.setv(dd, 2, 4) // set its value content to an existing double array of size 2X4

Mat(nrows: Integer, ncols: Integer, initValue: Double) // Creates a Mat with *nrows* rows, ncols columns, initialized to initValue

var m=new Mat(3, 4, 3.4)

// construct a Mat from a two-dimensional double array without copying the array values

Mat(da: Array[Array[Double]], flag: Boolean) // Creates a Mat initialized with a reference to the da array

// construct a Mat from an one-dimensional double array

Mat(da: Array[Double])

var od = new Array[Double](10)

od(2)=2.2

var odm = new Mat(od)

Basic Methods

def size: (Int, Int) // Returns the number of rows and columns of the Mat

var md = ones0(9)

md.size

def getv: Array[Array[Double]] *//* returns the Java 1-D double array used to store the contents of the matrix

def length: Int // Returns the number of rows of Mat

var md = ones0(9, 2)

md.length

def Nrows: Int // Returns the number of rows of Mat

md.Nrows

def Ncols: Int // Returns the number of columns of Mat

md.Ncols

def toDoubleArray(): Array[Array[Double]] // a very useful method that returns the flat EJML one

// dimensional storage, as a two-dimensional Java array

e.g.

val A = ScalaSci.EJML.StaticMathsEJML.rand0(5,6)

val Ad = A.toDoubleArray

val Ad2 = ScalaSci.EJML.StaticMathsEJML.toDoubleArray(A)

Some auxiliary routines

def ones0(n: Int): Mat - constructs a nXn Mat of ones

def ones0(n: Int, m: Int): Mat - constructs a nXm Mat of ones

def zeros0(n: Int): Mat - constructs a nXn Mat of zeros

def zeros0(n: Int, m: Int): Mat - constructs a nXm Mat of zeros

def rand0(n: Int): Mat - constructs a nXn Mat of random values

def rand0(n: Int, m: Int): Mat - constructs a nXm Mat of random values

var rr = rand0(5,8)

Apply Operators

var rr = rand0(5,8)

var rr01 = rr(0,1) // gets the corresponding double value

var r0 = rr(0) // extracts the 0^{th} row

Update Operators

rr(0,1) = 99 // sets the corresponding element

Transpose matrix

Transpose Matrix, operator ~, or method transpose

```
var mm = ones0(9,4) // create an EJML matrix of 1s
mm(2,3) = 23 // update one element
var mmt = mm~  // using the ~ operator for transposing
var mmt2 = mm.transpose // also transposes
var mmt3 = transpose(mm) // alternative interface

mm.print
mmt.print
```

Inverse matrix

var mm= rand0(8)

var mmi = inv(mm) // get the inverse

*var oned = mm*mmi // verify that indeed we get the inverse*

Retrieve lower-level EJML data structures

This is particularly useful when we have to work directly with EJML routines

We can retrieve the SimpleMatrix on which ScalaSci's Mat is based with:

var rnd = rand0(5,6) // a random EJML based matrix
var smat = rnd.sm

And also, the DenseMatrix64F on which SimpleMatrix is based:

var denseM = smat.getMatrix

Access operations

Access operations are implemented conveniently with a MATLAB-like style, e.g.

var a = rand0(10, 20)
var a1 = a(1) // get the 1th row as a matrix
var a1_3 = a(1::3, ::) // get rows 1 until 3 all columns
var ac2_5 = a(::, 2, 5) // get columns 2 until 5 all rows
var ac2_3_1_5 = a(2::3, 1::5) // get rows 2 until 3, columns 1 until 5

Operators

Matrix addition, subtraction and multiplication has been overloaded and works properly. Also addition and multiplication with scalars operates as expected, e.g.

var m1 = rand0(4,7)
var m2 = rand0(7, 9)
*var mmul = m1*m2 // multiplication*
*var madd = m1+7*m1+m1*9.4*
org.ejml.ops.MatrixVisualization.show(madd.sm.getMatrix, "madd matrix")

Some simple examples involving EJML: operations follow.

Inverse Matrix, inv()

var mm= rand0(8)
var mmi = inv(mm) // get the inverse
*var oned = mm*mmi // verify that indeed we get the inverse*
oned.print

Solve Linear System, solve()

var A = M("3 -1 1; -1 3 1; 1 1 3")
var b = M("3; -5; -4")
var x = A.solve(b) // solve thye system
*var bres = A*x // estimate the quality of the solution*
var diff = b-bres
diff.print // we should have zeros

Map a function to each element of the matrix

Let now transform all the elements of a matrix with the function f(x) = 2*x+10

```
def f(x: Double) = 2*x+10
var m = ones0(5,10) // a simple test matrix consisting of all ones
var mTrans = m map f // the transformed matrix
mTrans.print
```

Create a diagonal matrix (diag0)

```
var diagElems = List(0.5, 2.3, -3.4, 2.3) // specify with a list the diagonal elements
var mm = diag0(diagElems) // create the matrix
mm.print
```

Trigonometric functions

All the basic trigonometric functions, e.g. sin, cos, tan, asin, acos, atan, cosh, sinh, tanh etc. are applicable to matrices, e.g.

```
val aa = rand0(5,8)
var aas = sin(0.34*aa)+tan(aa*8.9)-3.4*tanh(0.23*aa)
aa.print
aas.print
```

Norm Operations

```
var zeroMatrix = new Mat(3,4)
var unzeroMatrix = M("0.2 1; -2 3; 6 5") // create the 2X3 matrix
var unzeroVector = M("0.3; 1; -2; 3; 4")
var squareMatrix = M("0.2 1; -2 3")
var vs = conditionP(squareMatrix, 1) // should be 7.69
var vsq = conditionP(unzeroMatrix, 1) // should be 3.43
var vt = unzeroMatrix~ // transpose matrix
var vtt = conditionP(vt,1) // should be 3.48
```

Covariance operations

```
// isValid returns: 0 = is valid, 1 = failed positive diagonal,
// 2 = failed on symmetry, 2 = failed on positive definite
var m = identity0(3)
m.print
// nothing is wrong with it, is a valid covariance matrix, return 0
isValid(m) // 0
// negative diagonal term
m(1,1) = -3
isValid(m) // 1
// not symmetric
m = identity0(3)
m(1, 0)=30
isValid(m) // 2
// not positive definite
m = identity0(3)
m(1, 2)= -400; m(2,1) = -400
isValid(m) // 3
```

Block Matrices

EJML implements algorithms that are defined in terms of matrix blocks in order to better exploit the caches of modern processors. The class *BMat* in ScalaLab wraps operations in those matrices

Constructors

BMat(N: Int, M: Int): Constructs a NXM block matrix, e.g.

```
var bm = new BMat(5,12)  // construct directly a BMat
var ejmlbm = new org.ejml.data.BlockMatrix64F(10, 20) // construct an EJML
BlockMatrix
var bm2 = new BMat(ejmlbm) // wrap a BMap around the EJML BlockMatrix64F
```

Apply operators

The selection of BMat elements is performed in a MATLAB-like style, e.g. :

```
var bmr = rand0b(30, 40) // get a random block matrix of 30 rows by 40 cols
var bmr3_4 = bmr(3,4) // get the corresponding element
bmr(3,4) *= 100  // multiply it by 100
bmr3_4 = bmr(3,4)  // take the upadated value
var brows3to5 = bmr(3::5, ::) // take rows 3 to 5 all columns
var brows3to5by2 = bmr(3::2 ::5,  ::) // take rows 3 to 5 by 2, all columns
var bcols3to6 = bmr(::, 3::6) // take columns 3 to 6, all rows
var bcols3to6by2 = bmr(::, 3::2 ::6) // take columns 3 to 6 by 2, all rows
var brows2to5by2cols3to9by3 = bmr(2::2 ::5, 3::3 ::9)  // extract the corresponding submatrix
```

Get the wrapped BlockMatrix64F matrix

It can be very useful to get the EJMLs BlockMatrix64F object on which BMat is based. This is accomplished with *getBM*, eg :

```
var ejmlBM64F = bmr.getBM
```

Apply a function to all the BMat's elements with map

One of the most useful functional style operation is to apply a function to all the elements f. This can be done with the *map* method, e.g.

```
var bmo = ones0b(20, 30) // a block matrix of ones
def f(x: Double): Double = x*x*x/1.5
bmo = (bmo map f) map sin // map the function f and the sin function
```

Arithmetic Operators and Trigonometric functions

Arithmetic operators on block matrices are overloaded and can be performed as usual. Also trigonometric functions work on block matrices, e.g.

```
var bm = ones0b(40, 30)
var bm2 = ones0b(30, 50)
var ff = 10+bm
```

```
var sinb = 10.2*sin(bm)
var mminusTimes = -bm*bm2
var sm = ones0b(25,25)
var mExpr = sm+0.23*sm*sm - sm*7.4 +cos(0.45*sm) // arbitrary expression
```

Chapter 6 MTJ Interfacing

The Matrix Toolkit for Java (MTJ) is an open source Java matrix library (https://github.com/fommil/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 LAPACK package which is a Java translation of the famous LAPACK package [15]. The Netlib API obtained from the open-source project *netlib-java* (https://github.com/fommil/netlib-java) provides a low-level interface to the Java LAPACK 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.

We should note at this point that although the JLAPACK is obtained from automatic translation of Fortran LAPACK to Java, JLAPACK routines are generally very efficient, especially for large matrices.

LAPACK on which much of the functionality of MTJ builds upon has three levels of routines [15]

•*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.

MTJ uses an object-oriented design for its Matrix classes. For example the *AbstractMatrix* is one of its basic base classes. The following 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. 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 and performing the eigendecomposition and then preparing the results with a convenient Scala tuple for output:

```
// 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 */
// return the results
```

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


  }
```

ScalaLab can utilize the MTJ (http://code.google.com/p/matrix-toolkits-java/) library upon which builds a MATLAB-like easy to use way of performing matrix operations. To work with the MTJ library more conveniently, you can open an MTJ based Scala interpreter from the pop-up menu of the ScalaLab console. It is important that the lower level routines of MTJ are all available, and we can call them, either when ScalaLab has not a higher-level equivalent or when we require maximum execution speed. Here we describe some aspects of working with the MTJ library presenting many examples.

A high-level interface to the MTJ library

The ScalaSci.MTJ.Mat class implements zero-indexed two-dimensional dense matrices in ScalaSci that are based upon the Matrix Toolkit for Java Library.

Constructors

Mat(rows: Integer, cols: Integer) // Creates an MTJ Mat of size rows, cols initialized to zeros

```
var m=new Mat(2, 3)
m.print // print the contents
```

Construction by specifying the initial elements, e.g.

```
var m = M0("4.5 5.6 -4; 4.5 3 -3.4") // space separated
or
var m = M0("4.5, 5.6, -4; 4.5, 3, -3.4") // comma separated
```

or

```
var m = Mat(2,3, 4.5, 5.6, -4, 4.5, 3, -3.4)
```

Construct Mat by copying the array values, *Mat(da: Array[Array[Double]])* , creates an MTJ Mat initialized with the da array

```
var dd = Array.ofDim[Double](2,4)
dd(1)(1)=11
```

var mdd = new Mat(dd)
mdd.print // print finally the MTJ Matrix

Retrieve lower-level MTJ data structures

This is particularly useful when we have to work directly with MTJ routines

We can retrieve the DenseMatrix on which ScalaSci's Mat is based with:

var rnd = rand0(5,6) // a random MTJ based matrix
var dmmat = rnd.getDM

Access operations

Access operations are implemented conveniently with a MATLAB-like style, e.g.

var a = rand0(10, 20)
var a1 = a(1) // get the 1th row as a matrix
var a1_3 = a(1::3, ::) // get rows 1 until 3 all columns
var ac2_5 = a(::, 2::5) // get columns 2 until 5 all rows
var ac2_3_1_5 = a(2::3, 1::5) // get rows 2 until 3, columns 1 until 5

Operators

Matrix addition, subtraction and multiplication has been overloaded and works properly. Also addition and multiplication with scalars operates as expected, e.g.

var m1 = rand0(4,7)
var m2 = rand0(7,9)
*var mmul = m1*m2 // multiplication*
*var madd = m1+7*m1+m1*9.4*

Basic Methods

def size: (Int, Int) // Returns the number of rows and columns of the Mat

var md = ones0(9)
md.size

def length: Int // Returns the number of rows of Mat

var md = ones0(9, 2)
md.length

def Nrows: Int // Returns the number of rows of Mat

def Ncols: Int // Returns the number of columns of Mat

md.Nrows
md.Ncols

Routines

def ones0(n: Int): Mat - constructs a nXn Mat of ones

def ones0(n: Int, m: Int): Mat - constructs a nXm Mat of ones

def zeros0(n: Int): Mat - constructs a nXn Mat of zeros

def zeros0(n: Int, m: Int): Mat - constructs a nXm Mat of zeros

def rand0(n: Int): Mat - constructs a nXn Mat of random values

def rand0(n: Int, m: Int): Mat - constructs a nXm Mat of random values

Transpose Matrix, operator ~, or method transpose

```
var mm = ones0(9,4) // create an MTJ matrix of 1s
mm(2,3) = 23 // update one element
var mmt = mm~  // transpose the matrix
mm.print
mmt.print
```

Trigonometric functions

All the basic trigonometric functions, e.g. sin, cos, tan, asin, acos, atan, cosh, sinh, tanh etc. are applicable to matrices, e.g.

```
val aa = rand0(5,8)
var aas = sin(0.34*aa)+tan(aa*8.9)-3.4*tanh(0.23*aa)
aa.print
aas.print
```

Chapter 7 Native BLAS in ScalaLab

jblas (http://jblas.org/) is a fast linear algebra library for Java. It is designed by Mikio Braun. It is based on BLAS and LAPACK, the de-facto industry standard for matrix computations, and uses state-of-the-art implementations like ATLAS for all its computational routines, making it very fast. It is essentially a lightweight wrapper around the BLAS and LAPACK routines. These packages originated in the Fortran community, which explains their archaic API. jblas aims to make this functionality available to Java programmers such that they do not have to worry about writing JNI interfaces and calling conventions of Fortran code.

ScalaLab started experimentally to interface with these routines with its ***ScalaSci.JBLAS.Mat*** and ScalaSci***.JBLAS.StaticMathsJBLAS*** classes.

You can switch to the experimental JBLAS based Scala Interpreter from the **Configuration** menu

Then you execute for example the code:

var m1 = ones0(1000, 1100)

var m2 = ones0(1100, 1500)

tic

*var m12 = m1*m2*

var tmMul = toc

If you use also Scala Interpreters based on pure Java libraries to execute the same code, e.g. the EJML one, you should observe that matrix multiplication is significantly faster, i.e. about 4 to 6 times faster, using the Native BLAS.

ScalaLab exploits the efficient JBLAS matrix multiplication and combines it with Java multiplication in order to implement very fast matrix multiplication for its basic *RichDouble2DArray* matrix.

The **RichDouble2DArray** is the fundamental ScalaLab array type that is interoperable easily with Java/Scala Array[Array[Double]] type. We started to explore the Native BLAS for speeding some operations. Linear time operations as addition/subtraction and multiplication with a scalar cannot be

improved with native code, since the copy of data between JVM and native memory areas also requires linear time.

Also, we have implemented and eigenvalue computation routines using Native BLAS but in this case the speedup is not important in comparison to the Java based eigenvalue computation. The following script demonstrates these routines.

// benchmark eigendecompositions for RichDouble2DArray

var m = ones(140,140)

var N = 10 // repetitions

var g = bench(N, {()=> var evs = eig(m)}) // using default

var gfevals = bench(N, {()=> var eigvals = eigVals(m)}) // using native BLAS eigenvalues

var gfevecs = bench(N, {()=> var eigvecs = eigVecs(m)}) // using native BLAS eigenvectors

Support for JBLAS both with RichDouble2DArray and JBLAS Matrix type

JBLAS can improve significantly many common operations. The following script illustrates significant performance difference between Java methods and native ones.

var N=300 // a size of the matrix
var x = rand(N, N) // constructs a NXN RichDouble2DArray

// compute the eigendecomposition using Java
tic
var (evecs, evals) = eig(x)
var tmeigJ = toc

// compute the eigendecomposition using fast JBLAS routines
tic

```
var (fevals, fevecs) = x.feig
var tmeigJBLAS  = toc

// compute the SVD
tic
var svdJ = svd(x)
var tmSVDJava = toc

// compute the SVD using fast JBLAS routines
tic
var svdJBLAS = x.ffullSVD
var tmsvdJBLAS  = toc
```

The fast JBLAS routines for the **RichDouble2DArray** type are described below together with the code. Similar is the interface for the routines that can be called on **JBLAS** Matrix objects.

```
// Fast JBLAS based routines

 // compute the eigenvalues/eigenvectors using JBLAS
 def feig() = {
    val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
    val eigVals = org.jblas.Eigen.eigenvalues(dm)  // eigenvalues
    val eigVecs = org.jblas.Eigen.eigenvectors(dm)   // eigenvectors

    (eigVals, eigVecs)
    }

 // compute the  eigenvalues using JBLAS
 def feigenvalues() = {
   val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
   org.jblas.Eigen.eigenvalues(dm)
   }
```

```
// compute the eigenvectors using JBLAS
def feigenvectors() = {
  val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
  org.jblas.Eigen.eigenvectors(dm)
  }

// compute the symmetric eigenvalues using JBLAS
def fsymmetricEigenvalues() = {
  val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
  org.jblas.Eigen.symmetricEigenvalues(dm)
  }

// compute the symmetric eigenvectors using JBLAS
def fsymmetricEigenvectors() = {
  val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
  org.jblas.Eigen.symmetricEigenvectors(dm)
  }

//  Computes generalized eigenvalues of the problem A x = L B x.
// @param A symmetric Matrix A. Only the upper triangle will be considered. Refers to this
//  @param B symmetric Matrix B. Only the upper triangle will be considered.
//  @return a vector of eigenvalues L.
def fsymmetricGeneralizedEigenvalues(B: Array[Array[Double]]) = {
  val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
  org.jblas.Eigen.symmetricGeneralizedEigenvalues(dm, new DoubleMatrix(B))
  }


 /**
   * Solve a general problem A x = L B x.
   *
   * @param A symmetric matrix A, refers to this
   * @param B symmetric matrix B
   * @return an array of matrices of length two. The first one is an array of the eigenvectors X
   *         The second one is A vector containing the corresponding eigenvalues L.
```

```
    */
def fsymmetricGeneralizedEigenvectors(B: Array[Array[Double]]) = {
   val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
   org.jblas.Eigen.symmetricGeneralizedEigenvectors(dm, new DoubleMatrix(B))
  }

   /**
    * Compute Cholesky decomposition of A ( this )
    * @param A should be symmetric, positive definite matrix (only upper half is used)
    * @return upper triangular matrix U such that  A = U' * U
    */
 def fcholesky() = {
  val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
  org.jblas.Decompose.cholesky(dm)
  }


/** Solves the linear equation A*X = B , A is this */
def fsolve(B: Array[Array[Double]]) = {
  val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
  org.jblas.Solve.solve(dm, new DoubleMatrix(B))
  }


/** Solves the linear equation A*X = B for symmetric A, A is this */
def fsolveSymmetric(B: Array[Array[Double]]) = {
   val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
   org.jblas.Solve.solveSymmetric(dm, new DoubleMatrix(B))
  }

/** Solves the linear equation A*X = B for symmetric and positive definite A */
def fsolvePositive(B: Array[Double]) = {
  val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
  org.jblas.Solve.solvePositive(dm, new DoubleMatrix(B))
  }
```

```
 /**
    * Compute a singular-value decomposition of A, A is this
    *
    * @return A DoubleMatrix[3] array of U, S, V such that A = U * diag(S) * V'
    */
def ffullSVD() = {
  val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
  org.jblas.Singular.fullSVD(dm)
  }

    /**
    * Compute a singular-value decomposition of A (sparse variant), A is this
    * Sparse means that the matrices U and V are not square but
    * only have as many columns (or rows) as possible.
    *
    * @return A DoubleMatrix[3] array of U, S, V such that A = U * diag(S) * V'
    */
def fsparseSVD() = {
  val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
  org.jblas.Singular.sparseSVD(dm)
  }

def fsparseSVD( Aimag: Array[Array[Double]]) =  {
  val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
  org.jblas.Singular.sparseSVD(new org.jblas.ComplexDoubleMatrix(dm, new
DoubleMatrix(Aimag)))
  }


  /**
    * Compute the singular values of a matrix.
    *
    * @param A DoubleMatrix of dimension m * n, A is this
```

```
    * @return A min(m, n) vector of singular values.
    */
def fSPDValues() = {
  val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
  org.jblas.Singular.SVDValues(dm)
  }


   /**
    * Compute the singular values of a complex matrix.
    *
    * @param Areal, Aimag :Areal is this,  the real and imaginary components of a
ComplexDoubleMatrix of dimension m * n
    * @return A real-valued (!) min(m, n) vector of singular values.
    */
 def fSPDValues(B: Array[Array[Double]]) = {
   val dm = new DoubleMatrix(this.v)   // convert to JBLAS DoubleMatrix
   org.jblas.Singular.SVDValues(new ComplexDoubleMatrix(dm, new DoubleMatrix(B)))
        }



}
}
```

JBLAS fast routines

The routines that have been used above to implement Scala interfaces are based on JBLAS Java routines that we describe below.

JBLAS offers some useful LAPACK routines that are highly optimized and therefore for large matrices can run significantly faster than Java ones. This routines can be utilized from ScalaLab either with *Array[Array[Double]]* or with RichDouble2DArray types. These routines are provided with the class: **ScalaSci.JBLAS.JBLASNativeJavaInterface**:

```
package ScalaSci.JBLAS;

import org.jblas.ComplexDoubleMatrix;
import org.jblas.DoubleMatrix;

public class JBLASNativeJavaInterface {

   // Computes the eigenvalues of a general matrix.
public static ComplexDoubleMatrix jblas_eigenvalues(double [][]dM) {
   return org.jblas.Eigen.eigenvalues(new DoubleMatrix(dM));
}



  // Computes the eigenvalues and eigenvectors of a general matrix.
  // returns an array of ComplexDoubleMatrix objects containing the eigenvectors
  //        stored as the columns of the first matrix, and the eigenvalues as the
  //       diagonal elements of the second matrix.
public static ComplexDoubleMatrix[]  jblas_eigenvectors(double [][]dM) {
   return org.jblas.Eigen.eigenvectors(new DoubleMatrix(dM));
}

// Compute the eigenvalues for a symmetric matrix.
public static DoubleMatrix  jblas_symmetricEigenvalues(double [][]dM) {
   return org.jblas.Eigen.symmetricEigenvalues(new DoubleMatrix(dM));
}


// Computes the eigenvalues and eigenvectors for a symmetric matrix.
// returns an array of DoubleMatrix objects containing the eigenvectors
//       stored as the columns of the first matrix, and the eigenvalues as
//       diagonal elements of the second matrix.
public static DoubleMatrix []  jblas_symmetricEigenvectors(double [][]dM) {
   return org.jblas.Eigen.symmetricEigenvectors(new DoubleMatrix(dM));
}
```

```
// Computes generalized eigenvalues of the problem A x = L B x.
// @param A symmetric Matrix A. Only the upper triangle will be considered.
// @param B symmetric Matrix B. Only the upper triangle will be considered.
// @return a vector of eigenvalues L.

public static DoubleMatrix jblas_symmetricGeneralizedEigenvalues( double [][] A, double [][] B)
{
   return org.jblas.Eigen.symmetricGeneralizedEigenvalues(new DoubleMatrix(A), new
DoubleMatrix(B));
}

   /**
    * Solve a general problem A x = L B x.
    *
    * @param A symmetric matrix A
    * @param B symmetric matrix B
    * @return an array of matrices of length two. The first one is an array of the eigenvectors X
    *         The second one is A vector containing the corresponding eigenvalues L.
    */
public static DoubleMatrix [] jblas_symmetricGeneralizedEigenvectors( double [][] A, double [][]
B) {
   return org.jblas.Eigen.symmetricGeneralizedEigenvectors(new DoubleMatrix(A), new
DoubleMatrix(B));
}

   /**
    * Compute Cholesky decomposition of A
    *
    * @param A symmetric, positive definite matrix (only upper half is used)
    * @return upper triangular matrix U such that  A = U' * U
    */
public static  DoubleMatrix  jblas_cholesky(double [][]A) {
  return org.jblas.Decompose.cholesky(new DoubleMatrix(A));
}
```

```
/** Solves the linear equation A*X = B. */
public static DoubleMatrix jblas_solve(double [][]A, double [][] B) {
   return org.jblas.Solve.solve(new DoubleMatrix(A),  new DoubleMatrix(B));
}

/** Solves the linear equation A*X = B for symmetric A. */
public static DoubleMatrix jblas_solveSymmetric(double [][]A, double [][] B) {
   return org.jblas.Solve.solveSymmetric(new DoubleMatrix(A),  new DoubleMatrix(B));
}

/** Solves the linear equation A*X = B for symmetric and positive definite A. */
public static DoubleMatrix jblas_solvePositive(double [][]A, double [][] B) {
   return org.jblas.Solve.solvePositive(new DoubleMatrix(A),  new DoubleMatrix(B));
}

 /**
    * Compute a singular-value decomposition of A.
    *
    * @return A DoubleMatrix[3] array of U, S, V such that A = U * diag(S) * V'
    */

public static DoubleMatrix []  jblas_fullSVD( double [][]A) {
   return org.jblas.Singular.fullSVD(new DoubleMatrix(A));
}


   /**
    * Compute a singular-value decomposition of A (sparse variant).
    * Sparse means that the matrices U and V are not square but
    * only have as many columns (or rows) as possible.
    *
    * @param A
    * @return A DoubleMatrix[3] array of U, S, V such that A = U * diag(S) * V'
    */
```

```
public static DoubleMatrix []  jblas_sparseSVD( double [][]A) {
   return org.jblas.Singular.sparseSVD(new DoubleMatrix(A));
}


public static ComplexDoubleMatrix []  jblas_sparseSVD( double [][]Areal, double [][] Aimag) {
   return org.jblas.Singular.sparseSVD(
         new ComplexDoubleMatrix(new DoubleMatrix(Areal),  new DoubleMatrix(Aimag)));
}


 /**
    * Compute the singular values of a matrix.
    *
    * @param A DoubleMatrix of dimension m * n
    * @return A min(m, n) vector of singular values.
    */

public static DoubleMatrix jblas_SPDValues(double [][]A) {
   return  org.jblas.Singular.SVDValues(new DoubleMatrix(A));
}


   /**
    * Compute the singular values of a complex matrix.
    *
    * @param Areal, Aimag : the real and imaginary components of a  ComplexDoubleMatrix of
dimension m * n
    * @return A real-valued (!) min(m, n) vector of singular values.
    */

public static DoubleMatrix jblas_SPDValues(double [][]Areal, double [][]Aimag) {
   return  org.jblas.Singular.SVDValues(
         new ComplexDoubleMatrix(new DoubleMatrix(Areal), new DoubleMatrix(Aimag)));
}
```

```
}
```

An example of using them follows:

```
var A = AAD("3.4 5.6  -3.4; 0.45 0.545 -1.3; 5.3 5.9 -2.3")  // create a RichDouble2DArray

// compute the eigenvalues using JBLAS
var JBLASeigs = jblas_eigenvalues(A)

// compute the eigenvectors using JBLAS
var JBLASeigsevecs = jblas_eigenvectors(A)

// compute the eigenvalues/eigenvectors  using EJML
var ejmleigs = ScalaSci.EJML.StaticMathsEJML.eig(A)

// create a (large) symmetric matrix
var N = 200
var Mdd = Array.ofDim[Double](N, N)
for (r<-0 until N)
 for (c<-0 until N) {
   var denom = 1.0+r*c
   Mdd(r)(c) = 1.0/ denom
}

// compute with LAPACK specific routine for symmetric matrices the eigenvalues
tic; var JBLASsym = jblas_symmetricEigenvectors(Mdd); var tmJBLASsym = toc

// compute with LAPACK general routine
tic; var JBLASgen = jblas_eigenvectors(Mdd); var tmJBLASgen = toc

// compute with EJML library
```

tic; var EJMLeigs= ScalaSci.EJML.StaticMathsEJML.eig(Mdd); var tmEJML= toc

Chapter 8 Plotting

ScalaLab implements a plotting interface similar to MATLAB. The low-level functionality is built upon the *jmathPlot* library (https://github.com/yannrichet/jmathplot). This chapter describes the architecture of the plotting system and some useful routines.

The default plotting system based on the jmathPlot library

The figure system is implemented by using an array of *FrameView* objects to keep the set of active figure frame. Also, a 3-D array of *PlotPanel* objects keeps references to the subplots, i.e. each FrameView maps to a 2-D array of PlotPanel objects that keep the subplots of that figure frame. The basic plotting routines are as follows:

figure(figId: Int)

Creates a figure window.

figure, by itself, creates a new figure window, and returns its handle.

figure(H) makes H the current figure, forces it to become visible, and raises it above all other figures on the screen. If Figure H does not exist, and H is an integer, a new figure is created with handle H.

subplot(m: Int, n: Int, p: Int)

Create axes in tiled positions.

subplot(m,n,p), breaks the Figure window into an m-by-n matrix of small axes, selects the p-th axes for the current plot, and returns the axis handle. The axes are counted along the top row of the Figure window, then the second row, etc. For example,

var t = 0::0.01::10

*var x = sin(0.13*t); var y= cos(3.4*t)*

subplot(2,1,1); plot(x);

subplot(2,1,2); plot(y)

plots **x** on the top half of the window and **y** on the bottom half.

closeAll, close("all")

closes all figures

close(fig: Int)

closes the figure with identifier fig

var f = figure

constructs a new 2-D figure and returns its integer handle

var f3 = figure3d

constructs a new 3-D figure and returns its integer handle

var plPanel = figure(figId: Int): ScalaSci.math.plot.PlotPanel

focuses on figure with identifier figId if it exists,

otherwise constructs this object

returns the PlotPanel object of figure figId

var plPanel = figure3d(figId: Int): ScalaSci.math.plot.PlotPanel

focuses on figure with identifier figId if it exists,

otherwise constructs this object

returns the PlotPanel object of figure figId

hold("on"), hold("off"), hold(true), hold(false)

```
var N=pow(2, 15).toInt; var t=linspace(0, 10, N);
var (noise, sig) = (vrand(N), sin(2.3*t))
figure(1);  plot(t, sig, Color.BLUE, "signal"); hold("on"); plot(t, noise, Color.RED, "noise");
```

clf(figId: Int)

clears the contents of figure with identifier figId

Example

```
closeAll
var N = pow(2, 12).toInt // a power of 2 integer
var t = linspace(0, 2, N) // get N samples from 0 to 2 with linear sampling
var (f1, f2, f3) = (2.23, 4.5, 9.8) // some frequencies for signals
var sig = sin(f1*t)+4.5*cos(f2*t)+1.9*sin(f3*t)
var x1 = vrand(N)+sig // add some noisy
var fftX1 = ScalaSci.math.array.BasicDSPSimple.fft(x1) // perform FFT
subplot(2,1,1); plot(t, x1, "signal")
subplot(2,1,2); plot( fftX1, "ffT")
f2 = getDouble("Get new signal frequency", 8.0)  // a dialog to read frequency from the user,
                                                 // by default 8.0 is used
```

```
sig = sin(f2*t)
var fftX2 = ScalaSci.math.array.BasicDSPSimple.fft(sig)
clf(1, 1, 2) // clears all plots from the second subplot of figure 1
subplot(2, 1, 2)
plot(fftX2, "fft for the new frequency = "+f2)
clf(1, 1, 1)
subplot(2, 1, 1)
plot(sig); title("the new signal")
```

Marked Line Plots

// display a plot with 'X' connected by a line

```
var t= 0::0.1::10
var x = sin(0.23*t)+0.2252*cos(3.4*t)

plotMarkedLine(t,x, Color.GREEN)
```

Plotting Examples

We present here some examples of using the default jmathPlot based ScalaLab plotting library.

Listing 9 Illustration of plots using JmathPlot library

```
var t = 0::0.01::20
var x = sin(0.2*t)
figure(1); title("Demonstrating ploting multiple plots at the same figure")
plot(t, x, Color.GREEN, "sin(0.2*t)")
// the xlabel(), ylabel() here refer to the axis of the current plot (i.e. PlotPanel object)
xlabel("t-Time axis")
ylabel("y=f(x) axis")
var  y = sin(0.2*t)+5*cos(0.23*t)
hold("on")
plot(t,y, new Color(0, 0, 30), "sin(0.2*t)+5*cos(0.23*t)" )
```

```
t = 0::0.01::20
y = sin(0.32*t)+5*cos(0.23*t)
var z = sin(1.2*t)+0.5*cos(0.23*t)
var fig = figure3d(2)
plot(t,y, z,  Color.BLUE, "Ploting in 3-D")
// specify labels explicitly for the fig PlotPanel object
fig.xlabel("t - Time axis ")
fig.ylabel("y - sin(0.2*t)+5*cos(0.23*t)")
fig.zlabel("sin(1.2*t)+0.5*cos(0.23*t);")
title("A 3-D plot");


var zcross = z cross z   // take the cross product
fig = figure3d(3)  // the 3nd figure handle
plot(t,y, zcross)
// specify labels explicitly for the fig PlotPanel object
xlabel("x - Time axis ")
ylabel("y - sin(0.2*t)+5*cos(0.23*t)")
zlabel("(sin(1.2*t)+0.5*cos(0.23*t)) .* (sin(1.2*t)+0.5*cos(0.23*t))")

var zcross2 = zcross cross  zcross  // again a cross product
fig = figure3d(4)  //  the 4th figure handle
subplot3d(2,1,1)
plot(t,y, zcross, Color.YELLOW)
// specify labels explicitly for the fig PlotPanel object
xlabel("x - Time axis ");
ylabel("y - sin(0.2*t)+5*cos(0.23*t)")
zlabel("(sin(1.2*t)+0.5*cos(0.23*t)) .* (sin(1.2*t)+0.5*cos(0.23*t))")
subplot3d(2,1,2)
plot(t,y, zcross, Color.MAGENTA)

// specify labels explicitly for the fig PlotPanel object
xlabel("x - Time axis ")
ylabel("y - sin(0.2*t)+5*cos(0.23*t)")
```

```
zlabel("zcross")
```

Listing 10 Illustration of jMathPlot based ScalaLab plots

```
var  t = 0::0.01::20
var  x = sin(0.2*t)
var  y = sin(0.2*t)+5*cos(0.23*t)
var  z = exp(-0.2*y)
figure(1)
hold("on")
plot(t, x, Color.RED, "sin(0.2*t)")
plot(t, y, Color.BLUE, "sin(0.2*t)+5*cos(0.23*t)")
plot(t, z, Color.GREEN, "exp(-0.2*y)")
xlabel("x-Time axis")
ylabel("x-y-z plots")
title("With hold on")

figure(2)
plot(t,x)
hold("off")
plot(t, y)
xlabel("x-Time axis")
ylabel("y-sin(0.2*t)")
title("With hold off")

figure(3);
plot(t, x, Color.RED)
hold("on");
plot(t, y, Color.BLUE)
xlabel("x-Time axis")
ylabel("y-sin(0.2*t)")
title("With hold on")
```

```
        var fig = figure(4)
// specify labels explicitly for the fig PlotPanel object
        fig.xlabel("x - Time axis ")
        fig.ylabel("y - sin(0.2*t)+5*cos(0.23*t)")
        x = t
            y = sin(0.2*t)+5*cos(0.23*t)
            plot(x,y)

        z = sin(1.2*t)+0.5*cos(0.23*t)
        fig = figure3d(5)
        plot(t, y,  z, Color.GREEN)
// specify labels explicitly for the fig PlotPanel object
         fig.xlabel("x - Time axis  t");
         fig.ylabel("y = sin(0.2*t)+5*cos(0.23*t)")
         fig.zlabel("z = sin(1.2*t)+0.5*cos(0.23*t)")



         var  zcross = z cross z
         fig = figure3d(6)
         plot(t, y,  zcross)
// specify labels explicitly for the fig PlotPanel object
         xlabel("x - Time axis ")
         ylabel("y - sin(0.2*t)+5*cos(0.23*t)")
         zlabel("(sin(1.2*t)+0.5*cos(0.23*t)) .* (sin(1.2*t)+0.5*cos(0.23*t))")

         var zcross2 =  zcross cross zcross
         fig = figure3d(7)
         subplot3d(2,1,1)
         plot(t, y, zcross)
// specify labels explicitly for the fig PlotPanel object
         xlabel("x - Time axis ")
         ylabel("y - sin(0.2*t)+5*cos(0.23*t)")
         zlabel("(sin(1.2*t)+0.5*cos(0.23*t)) .* (sin(1.2*t)+0.5*cos(0.23*t))")

        subplot3d(2,1,2)
```

```
      plot(t, y, zcross)

// specify labels explicitly for the fig PlotPanel object
       xlabel("x - Time axis ")
       ylabel("y - sin(0.2*t)+5*cos(0.23*t)")
       zlabel("zcross")
```

Listing 11 Illustration of jMathPlot based ScalaLab plots

```
var  dt = 0.01  // sampling frequency
var xl = -10;  var  xu = 20;  // low and up limits
var t=xl::dt::xu   // time axis
var f11 = 0.23;  var f12 = 3.7  // two frequencies
var f21 = 0.25;  var f22 = 3.9  // slightly different frequencies

var  x1 = sin(f11*t) + cos(f12*t)
var  x2 = sin(f21*t) + cos(f22*t)
figure(1);  hold("on"); plot(t, x1, Color.RED, "1st sine")
plot(t, x2, Color.GREEN, "2nd sine")
var x12 = x1 dot  x2
plot(t, x12, Color.BLUE, "x1 .* x2")
```

Listing 12 Illustration of jMathPlot based ScalaLab plots

```
// plots a sigmoid
var a = 1.7159
var b=2.0/3.0
var u = -2::0.001::2

var fu = a*tanh(b*u)
for (k<-0 to 20) {
 if (k<5)
   plot(u, fu, new Color(k*20, 0, 0));
 else {
 if (k<10)
```

```
  plot(u, fu, new Color(100, k*20, 0));
 else
  plot(u, fu, new Color(0, 10*k, k*10));
}

hold("on")
  b=1.4*b
  fu = a*tanh(b*u)
 }
```

Listing 13 Illustration of jMathPlot based ScalaLab plots

```
// display a plot with  'X' connected by a line
var t= inc(0, 0.1, 10);
var x = sin(0.23*t)+0.2252*cos(3.4*t)

plotMarkedLine(t,x, Color.GREEN)
```

Listing 14 Illustration of jMathPlot based ScalaLab plots

```
 var p = new Plot3DPanel();

 //triangular random cloud (as sum of two uniform random numbers)
 var cloud = Array.ofDim[Double](1000,3);
 for ( i <- 0 to cloud.length-1) {
            cloud(i)(0) = Math.random + Math.random;
            cloud(i)(1) = Math.random + Math.random;
            cloud(i)(2) = Math.random + Math.random;
              }
   p.addCloudPlot("cloud", Color.RED, cloud, 3, 3, 3);

  var  cloud2 = Array.ofDim[Double](1000,3);
  for ( i<- 0 to cloud.length-1) {
```

```
        cloud2(i)(0) = 2 + Math.random + Math.random;
        cloud2(i)(1) = 2 + Math.random + Math.random;
        cloud2(i)(2) = 2 + Math.random + Math.random;
    }
 p.addCloudPlot("cloud2", Color.GREEN, cloud2, 3, 3, 3);
 p.setLegendOrientation(PlotPanel.SOUTH);
 new FrameView(p)
```

Listing 15 Illustration of jMathPlot based ScalaLab plots

```
var  p2 = new Plot2DPanel;
var XYZ = Array.ofDim[Double](10, 2);
var sXYZ = new Array[Double](10);
  for (i <- 0 to 1) {
  for (j <- 0 to XYZ.length-1) {
     XYZ(j)(0) = Math.random;
        XYZ(j)(1) = Math.random;
        sXYZ(j) = Math.random;
    }
  p2.addScatterPlot("toto" + i, XYZ);
 }
                    p2.getPlot(0).addGaussQuantiles(1, sXYZ);
                    p2.getPlot(1).addGaussQuantiles(1, 0.1);

                    new FrameView(p2)
```

Listing 16 Illustration of jMathPlot based ScalaLab plots

```
 var  X = new Array[Double](500);
 for (i <- 0 to X.length-1)
        X(i) = Math.random+Math.random;
```

```
var p = new Plot2DPanel("SOUTH");
p.addHistogramPlot("test", X, 10);
new FrameView(p);
```

Listing 17 Illustration of jMathPlot based ScalaLab plots

```
var  XY = Array.ofDim[Double](500,2);
for ( i <-  0 to XY.length-1) {
        XY(i)(0) = Math.random+Math.random;
        XY(i)(1) = Math.random+Math.random;
        }
        var p = new Plot3DPanel("SOUTH");
        p.addHistogramPlot("test", XY, 4, 6);
        new FrameView(p);
```

Listing 18 Illustration of jMathPlot based ScalaLab plots

```
closeAll  // closes any open figure

var n=100
var m = 50
var X = new Array[Double](n)
var Y = new Array[Double](m)
var Z1 = Array.ofDim[Double](m, n)
var Z2 = Array.ofDim[Double](m, n)

var alpha1 = getDouble("Plotting exponential. Specify alpha in Y = exp(-alpha*(X+Y))",  4)
var alpha2 = getDouble("Plotting exponential. Specify alpha in Y = exp(-alpha*(X+Y))",  15)
var i=0; var j=0;
var xlen = X.length
var xlen2 = xlen/2.0
```

```
var ylen = Y.length
var ylen2 = ylen/2.0
while ( i < xlen) {
  X(i) = (i-xlen2)/xlen2
  j=0
  while (j<ylen) {
     Y(j)= (j-ylen2)/ylen2
     Z1(j)(i) = Math.exp(-alpha1*(X(i)+Y(j)))
     Z2(j)(i) = Math.exp(-alpha2*(X(i)+Y(j)))
     j += 1
     }
   i += 1
 }


  subplot3d(2,1,1); surf(X, Y, Z1, "alpha = "+alpha1)
subplot3d(2,1,2);  surf(X, Y, Z2, "alpha = "+alpha2)
```

Direct JMathPlots

The JMathPlot library can be used directly from ScalaLab. Here we present some examples adapted in Scala from jMathPlot of Yann Richet.

Line Plot Example

```
var x = Array(1,2,3,4,5,6.0) // X-values for plotting
var y = Array(45.0, 89, 6, 32, 63, 12)  // Y-values for plotting
var plot = new Plot2DPanel
plot.addLegend("SOUTH")
plot.addLinePlot("simple x-y plot", x, y)
var frame = new javax.swing.JFrame("a plot panel")
frame.setSize(600, 600)
frame.setContentPane(plot)
```

```
frame.setVisible(true)
```

Grid Plot Example

```
import java.lang.Math.PI
def fxy1(x: Double, y: Double) = cos(x*PI)*sin(y*PI)

def f1( x: Array[Double], y: Array[Double]) = {
  var z = Array.ofDim[Double](y.length, x.length)
  for (i<-0 until x.length)
   for (j<-0 until y.length)
   z(j)(i) = fxy1(x(i), y(j))
  z
  }

def fxy2(x: Double, y: Double) = sin(x*PI)*cos(y*PI)

def f2( x: Array[Double], y: Array[Double]) = {
  var z = Array.ofDim[Double](y.length, x.length)
  for (i<- 0 until x.length)
   for (j<-0 until y.length)
     z(j)(i) = fxy2(x(i), y(j))
 z
 }

 // define your data
 var x = Inc(0.0, 0.1, 1.0)
 var y = Inc(0.0, 0.05, 1.0)
 var z1 = f1(x, y)
 var z2 = f2(x, y)

 // create your PlotPanel
 var plot = new Plot3DPanel

 // add grid plot to the PlotPanel
plot.addGridPlot("z = cos(PI*x)*sin(PI*y)", x, y, z1)
plot.addGridPlot("z = sin(PI*x)*cos(PI*y)", x, y, z2)
 var frame = new javax.swing.JFrame("a plot panel")
 frame.setSize(800, 600)
 frame.setContentPane(plot)
 frame.setVisible(true)
```

Histogram

```
import ScalaSci.math.array.StatisticSample._
// define your data
var x = randomLogNormal(1000, 0, 0.5)  // 1000 random numbers from a log normal statistical law
// create your PlotPanel (you can use it as a JPanel)
var plot = new Plot2DPanel()
// add the histogram (50 slices)  of x to the PlotPanel
plot.addHistogramPlot("Log Normal population", x, 50)
var frame = new javax.swing.JFrame("a plot panel")
```

```
frame.setSize(600, 600)
frame.setContentPane(plot)
frame.setVisible(true)
```

Customize your plot

```
import _root_.java.awt._ ;
import _root_.java.awt.event._ ;
import _root_.javax.swing._ ;
import _root_.javax.swing.event._;
import ScalaSci.math.array.StatisticSample._
import ScalaSci.math.plot.plotObjects._
import java.lang.Math.PI
// 1000 random numbers from a normal (Gaussian) statistical law
var x = randomNormal(1000, 0, 1)
// 1000 random numbers from a uniform statistical law
var y = randomUniform(1000, -3, 3)
// create your PlotPanel (you can use it as a JPanel)
var plot = new Plot2DPanel
// legend at SOUTH
plot.addLegend("SOUTH")
// add the histogram (50 slices) of x to the PlotPanel
plot.addHistogramPlot("Gaussian population", x, 50)
// add the histogram (50 slices) of y to the PlotPanel in GREEN
plot.addHistogramPlot("Uniform population", Color.GREEN, y, 50)
// add a title
var title = new BaseLabel("... My nice plot ... ", Color.RED, 0.5, 1.1)
title.setFont(new Font("Courier", Font.BOLD, 20))
plot.addPlotable(title)
// change name of axis
plot.setAxisLabels("<X>", "frequency")
// customize X axis
// rotate light labels
plot.getAxis(0).setLightLabelAngle(-PI/4)
// change axis title position relatively to the base of the plot
plot.getAxis(0).setLabelPosition(0.5, -0.15)
// customize Y axis
// rotate light labels
plot.getAxis(0).setLightLabelAngle(-PI/4)
// change axis title position relatively to the base of the plot
plot.getAxis(0).setLabelPosition(-0.15, 0.5)
 // change axe title angle
plot.getAxis(1).setLabelAngle(-PI/2)
// put the PlotPanel in a JFrame like a JPanel
var frame = new javax.swing.JFrame("a plot panel")
frame.setSize(600, 600)
frame.setContentPane(plot)
frame.setVisible(true)
```

Chapter 9 Plots with the JFreeChart Plotting Library

The JFreeChart interface in ScalaLab

The *JFreeChart* plotting library is one of the most famous Java plotting libraries. This library is provided by default with ScalaLab and the .jar file of JFreeChart is whithin the ScalaLab's lib folder. Therefore, *JFreeChart* can be utilized either from *ScalaSci* or with Java code. However, the default interface is not very convenient; so ScalaLab implements an alternative MATLAB-like interface.

We should not as a general rule that plotting routines based on JFreeChart are similar to the default plotting routines but they start with j, e.g. *jsubplot* instead of *subplot.*

The figure system is implemented by keeping an array of *JFrame* objects to keep the set of active figure frames. Also, a 3-D array of *jPlot* objects keeps references to the subplots, i.e. each *JFrame* maps to a 2-D array of *jPlot* objects that keep the subplots of that figure frame.

It is important to note beforehand that the JFreeChart plotting imports are not by default importing. Thus we should append the following imports:

import _root_.JFplot._;

import _root_.JFplot.jFigure._;

The jPlot class

A basic component of the plotting machinery is the *jPlot* class. This class encapsulates JFreeChart's XY charts. It simplifies the utilization of JFreeChart by providing an interface similar to MATLAB's plot.

To create a chart, we need X and Y values stored in Array[Double]. This procedure is illustrated by the following example:

var x = Inc(0, 0.01, 10) // sample from 0 to 10 with 0.01 step, collecting values in the x array pf doubles

*var y = sin(0.23*x)+2.4*cos(0.87*x) // construct a sinusoidal function*

var jchart = new jPlot(); // create a chart object

jchart.jplot(x,y) // plot the function

jchart.showInNewFrame // show the plot in the internal JFrame that the jPlot class maintains

*jchart.setTitle("Plot of sin(0.23*x)+2.4*cos(0.87*x) ") // set a title for the plot*

jchart.setXLabel("x-axis") // label for x-axis

*jchart.setYLabel(" sin(0.23*x)+2.4*cos(0.87*x") // label for y-axis*

If we want to display a chart in an existing window, we can get a *JPanel* object that will contain the chart by calling the method *getPanel().*

For example we can place the chart that we constructed with the previous code in a new JFrame as:

var newFrame = new JFrame(" Another plot frame")

newFrame.add(jchart.getPanel()) // add our chart to the new window

newFrame.setSize(800, 800)

newFrame.setVisible(true)

We can configure chart properties (colors, axis ranges...) by setter methods. Most *set...()* methods may be called before or after *plot()* and *addPlot()* methods. The exception to this rule are methods, which set color and style of lines. They all have *lineIndex* or *lineId* as first parameter. These methods may be called only after the line with the given index or id has been drawn.

As in MATLAB we can set the “hold” state of a plot. When hold state is on, additional plots are drawn without erasing the existing plots. The hold state is adjusted by the *setHold(holdState: Boolean)* method, i.e.

jchart.setHold(true) // sets the hold state on

jchart.setHold(false) // sets the hold state off

Examples:

We draw a green, dashed line, with 'x' markers and legend 'temperature' to the chart of the previous example:

*var temperature = 0.8+ sin(0.9*x) // take as temperature an arbitrary function*

jchart.setHold(true) // adjust the hold state

var lineId = jchart.jplot(x, temperature, "g--x", "temperature")

Note that we used *lineId* to store id for use in the next statement, where we'll change the line style of the line with index 0, and color to red dotted line with diamond * markers. Width will be 5 pixels:

jchart.setLineSpec(lineId(0), "r:d", 5)

The same can be done by specifying the last plotted line (we change it to BLUE color):

jchart.setLineSpec(jPlot.LAST_IDX, "b:d", 5)

As we said, we can have more than one line on the chart at the same time. This can be achieved in one of three ways:

1. by specifying all lines in a single plot command, for example

var x = inc(0, 0.01, 10) // sample from 0 to 10 with 0.01 step, collecting values in a Vec x

*var y1 = sin(0.23*x)+2.4*cos(0.87*x) // construct a sinusoidal function*

*var y2 = cos(0.78*x)-7.8*sin(0.12*x) // a second sinusoidal signal*

var jchart = new jPlot(); // create a chart object

jchart.jplot(x, y1, x, y2) // plot all the Vectors

jchart.showInNewFrame // show the plot in the internal JFrame that the jPlot class maintains

jchart.setTitle("Multiple plots") // set a title for the plot

jchart.setXLabel("x-axis") // label for x-axis

jchart.setYLabel(" y-axis") // label for y-axis

2. by calling

jchart.setHold(true)

before calling next plot commands

3. by calling method *addPlot()* instead of *jplot(),* for example:

var x = inc(0, 0.01, 10) // sample from 0 to 10 with 0.01 step, collecting values in a Vec x

*var y1 = cos(0.1*x)-1.8*sin(0.12*x) // a second sinusoidal signal*

*var y2 = cos(0.78*x)-7.8*sin(0.12*x) // a second sinusoidal signal*

var jchart = new jPlot(); // create a chart object

jchart.jplot(x, y1) // plot the first line

var dummy = 0 // a dummy parameter for disambiguation of overloaded definitions

jchart.addPlot(x, y2, dummy) // add the second line

jchart.showInNewFrame // show the plot in the internal JFrame that the jPlot class maintains

jchart.setTitle("Multiple plots") // set a title for the plot

jchart.setXLabel("x-axis") // label for x-axis

jchart.setYLabel(" y-axis") // label for y-axis

Line specification

Methods *jplot()*, *addPlot()*, and *setLineSpec()* have parameter *lineSpec*, which defines line color, style, and markers. Any combination of color, marker and style is allowed. The following properties can be specified:

Colors:

r red

g green

b blue

c cyan

y yellow

m magenta

k black

Markers:

+ Plus sign

o Circle

* Asterisk

. Point

x Cross

^ Upward-pointing triangle

v Downward-pointing triangle

> Right-pointing triangle

< Left-pointing triangle

'square' or s Square

'diamond' or d Diamond

'pentagram' or p Five-pointed star (pentagram)

'hexagram' or h - Six-pointed star (hexagram)

Styles:

- solid line

– dashed line

: dotted line

-. dash-dot line

Examples:

"yx--" yellow dashed line with crosses at points

":c" dotted cyan line without markers

"w" white solid line (the same as "w-")

JFreeChart based Plotting routines

jfigure(figId: Int)

Create figure window.

figure, by itself, creates a new figure window, and returns its handle.

figure(H) makes H the current figure, forces it to become visible, and raises it above all other figures on the screen. If Figure H does not exist, and H is an integer, a new figure is created with handle H.

jsubplot(m: Int, n: Int, p: Int)

Create axes in tiled positions.

subplot(m,n,p), breaks the Figure window into an m-by-n matrix of small axes, selects the p-th axes for the current plot, and returns the axis handle. The axes are counted along the top row of the Figure window, then the second row, etc. For example,

var t = inc(0,0.01,10)

*var x = sin(0.13*t); var y= cos(3.4*t)*

jfigure(1)

jsubplot(2,1,1); jplot(t, x);

jsubplot(2,1,2); jplot(t, y)

plots **x** on the top half of the window and **y** on the bottom half.

JFreeChart plotting demo

jfigure(1)

*var t = inc(0, 0.01, 10); var x = sin(0.23*t)*

var lineSpecs = "."

jplot(t,x, lineSpecs)

jtitle("drawing multiple line styles")

```
jhold(true) // hold axis
var SX = 5 // subsample factor
var tsub = t(1, SX, t.length-SX) // subsampe
var xsub = x(1, SX, x.length-SX)
jplot(tsub, 0.4*xsub, lineSpecs)
lineSpecs = ":r+"
jplot(tsub, 0.1*xsub, lineSpecs)
// redefine the color of line 2
jlineColor(2, Color.BLUE)

jfigure(2)
jsubplot(222)
var x11 = sin(8.23*t)
jplot(t,x11)
jhold(true)
lineSpecs = ":g"
jplot(t,sin(5*x11), lineSpecs)
jsubplot(223)
lineSpecs = ":r"
jplot(t,x11, lineSpecs)


// create a new figure and preform a plot at subplot 3,2,1
var nf = jfigure
jsubplot(3,2,1)
var t2 = inc(0, 0.01, 10); var x2 = sin(3.23*t2)+2*cos(0.23*t2)
jplot(t2,x2, ".-")
jsubplot(3,2,3)
var x3 = cos(2.3*t2)+9*sin(4.5*t2)
jplot(t2, x3)
jlineColor(1, Color.RED)
jsubplot(3,2,5)
var x4 = cos(12.3*t2)+9*sin(2.5*t2)
jplot(t2, x4+x3)
jlineColor(1, Color.GREEN)
```

```
jsubplot(3,2,6)
jplot(t2, 6*x4+x3)
jtitle("6*x4+x3")
jlineColor(1, Color.BLUE)

// now plot again at figure 2
jfigure(2) // concetrate on figure 2
jsubplot(2, 2, 1)
var vr = vrand(2000)
jplot(vr)
jtitle("A Random Vector")
var td = t.getv // get time as Array[Double]
jsubplot(224)
jplot(td, sin(1.34*td).getv, td, sin(3.6*td).getv, td, (cos(7.8*td)+2.3*sin(3.2*td)).getv)
jtitle("Multiple Plots")



// demonstrate PieDataChart

var categories = Array("Class1", "Class2", "Class3")
var values = Array(5.7, 9.8, 3.9)
var pieChartName = "Test Pie Chart"
var myPie = jplot(pieChartName, categories, values)
```

Chapter 10 Advanced Characteristics of the plotting system

Functional Style Plotting

Functional programming opens new opportunities for effective plotting. The following example demonstrates that fact.

Example

```
 import  _root_.JFplot._;
 import _root_.JFplot.jFigure._;
// Demonstrates functional style plotting in ScalaLab

// define and plot the corresponding 1-D function
def  fx(x:Double) = { x - sin(3.7*x)*x }
var (low, high, color, linePlotting, numPoints) = (1.0, 10.0, Color.BLUE, true, 4000)  // limits of plotting
fplot(fx,  low, high, color, linePlotting, numPoints)
var pointSize = 20
var textX = (high-low)/2
var textY = high/2
setLatexColor(Color.RED)
latexLabel("x - x*sin(3.7*x)", pointSize, textX, textY)


// plot two functions
def f1x(x: Double) = { x * cos(2.3*x) }
def f2x(x: Double) = { x - 30*sin(0.12*x) }
figure(2)
fplot(f1x, 1, 20, Color.BLUE)
setLatexColor(Color.BLUE)
latexLabel( "x*cos(2.3*x)",   15, 2.0, 1.0)
hold("on")
fplot(f2x, 1, 20, Color.GREEN )
```

```
setLatexColor(Color.GREEN)
latexLabel( "x - 30*sin(0.12*x)",  15, 2.0, -2.0)

// define as String and plot the corresponding 1-D function
figure(3)
var  textOfFunction = getString("Specify function of x", "sin(3.3*x)+4.5*cos(2.8*x)")
splot(textOfFunction, 1, 5)

// plot with JFreeChart
 jsplot("sin(x*cos(5.6*x))", 1, 5)


 // surface plots
 def  fxy1(x:Double, y: Double) = { x*sin(x)+x*y*cos(y) }
 figure3d(4)
 fplot2d(fxy1,  1, 10, 1, 5, Color.BLUE, false )
 def fxy2(x: Double, y: Double) = { x*y*cos(x*y)-x*sin(x) }
 figure3d(5)
 fplot2d(fxy2, 1, 10, 1, 5, Color.RED, true)


// define directly as a String and plot the corresponding 1-D function
 figure3d(6); splot2d("x*y*Math.cos(x*y)", -4, 4, -10, 10)
```

Named Plots interface

The named parameters can simplify interfaces consisting of routines with many parameters as the plotting routines are. Here we provide examples on how to use the named plotting interfaces of ScalaLab.

Example 1

```
var t = 0::0.05::10
var x = sin(0.6*t)
var y = sin(2.6*t)
figure(1)
subplot(2,1,1)
nplot(xvec=t, yvec=x)
```

```
subplot(2,1,2)
nplot(xvec=t, yvec=x, lineWidth=8)

figure(2)
var td = Inc(0, 0.1, 30)
var xd = sin(0.3*td)
subplot(3,1,1)
nplot(xdd=td, ydd=xd, color=Color.BLUE)
figure(3)
subplot(3,1,1)
nplot(xvec=t, yvec=x, plotType=".")
subplot(3,1,2)
nplot(xvec=t, yvec=x, plotType="x")
// demonstrate that we can change the order of named parameters
subplot(3,1,3)
nplot(yvec=x, xvec=t, plotType="*")
figure(4)
nplot2(x=t, y=x, z = y)
```

Object-Oriented Plotting

Plotting routines tend to be complicated and can easily confuse the user. Therefore, we start to develop more convenient, object oriented plotting routines. These routines keep state information and permit the user to easily control the properties of the plots. A snapshot of such code is illustrated below.

Design of a Plot Facilitator Object

```
/* a class to plot in an object-oriented way
 we perform plotting in two steps:
   1. We specify the properties of our desired plot using the set*() routines
   2. We perform the plot using the simple mkPlot*() routines
*/

class PlotController() {

// these routines affect the properties of the AbstractDrawer class
// of the jmathPlot library, and can be used to change the defaults
// for external plotting routines that do not specify their own values
      // set drawing to continuous lines
   def setContinuousLine() = { ScalaSci.math.plot.render.AbstractDrawer.line_type =
ScalaSci.math.plot.render.AbstractDrawer.CONTINOUS_LINE }
      // set drawing to dotted lines
   def setDottedLine() = { ScalaSci.math.plot.render.AbstractDrawer.line_type =
ScalaSci.math.plot.render.AbstractDrawer.DOTTED_LINE }
      // set drawing to Round Dot
   def setRoundDot() = { ScalaSci.math.plot.render.AbstractDrawer.dot_type =
ScalaSci.math.plot.render.AbstractDrawer.ROUND_DOT }
      // set drawing to cross dot
   def setCrossDot() = { ScalaSci.math.plot.render.AbstractDrawer.dot_type =
ScalaSci.math.plot.render.AbstractDrawer.CROSS_DOT }
      // set font
   def setFont(font: java.awt.Font) = { ScalaSci.math.plot.PlotGlobals.defaultAbstractDrawerFont
```

```
= font }
      // set Color
   def setColor(color: java.awt.Color) =
{ ScalaSci.math.plot.PlotGlobals.defaultAbstractDrawerColor = color }
      // set line width
   def setlineWidth(lw: Int) = { ScalaSci.math.plot.render.AbstractDrawer.line_width = lw }

    // references to the data for plot
   var xData = new Array[Double](1)
   var yData = new Array[Double](1)
   var zData = new Array[Double](1)

    // adjust the data references for plotting to the actual data
   def setX(x: Array[Double]) = { xData = x }  // sets the x-data for plotting
   def setY(y: Array[Double]) = { yData = y }  // sets the y-data for plotting
   def setZ(z: Array[Double]) = { zData = z }  // sets the z-data for plotting

   def setX(x: ScalaSci.Vec) = { xData = x.getv}  // sets the x-data using Vec
   def setY(y: ScalaSci.Vec) = { yData = y.getv} // sets the y-data using Vec
   def setZ(z: ScalaSci.Vec) = { zData = z.getv} // sets the z-data using Vec



   var xlabelStr = "X-axis"
   def setxlabel(xl: String) = xlabelStr = xl

   var ylabelStr = "Y-axis"
   def setylabel(yl: String) = ylabelStr = yl

   var zlabelStr = "Z-axis"
   def setzlabel(zl: String) = zlabelStr = zl

   var plotTitle2D = "2-D Plot"
   def setplotTitle2D(plTitle: String) = { plotTitle2D = plTitle }
   var plotTitle3D = "3-D Plot"
   def setplotTitle3D(plTitle: String) = { plotTitle3D = plTitle }
```

```
   // perform the plot using the object's properties
   def mkplot() = {
      ScalaSci.math.plot.plotTypes.plot(xData, yData,
ScalaSci.math.plot.PlotGlobals.defaultAbstractDrawerColor, plotTitle2D)
      ScalaSci.math.plot.plot.xlabel(xlabelStr)
      ScalaSci.math.plot.plot.ylabel(ylabelStr)

      }

 // perform the plot using the object's properties
   def mkplot3D() = {

      ScalaSci.math.plot.plotTypes.plot(xData, yData, zData,
ScalaSci.math.plot.PlotGlobals.defaultAbstractDrawerColor,  plotTitle3D)
      ScalaSci.math.plot.plot.xlabel(xlabelStr)
      ScalaSci.math.plot.plot.ylabel(ylabelStr)
      ScalaSci.math.plot.plot.zlabel(zlabelStr)
      }
 }
```

Testing the Plot Controller Object

We can now test the plot controller object as:

```
// Illustrates how to  construct a PlotController object to facilitate the plotting
   var po = new PlotController()

   // construct some signals
   var x = 0::1 ::100
   var y1 = sin(0.45*x)
   var y2 = sin(0.778*x)+0.2*cos(3.4*x)
   var y3 = cos(y1+y2)

   // construct the first plot object
   closeAll()  // close any previous figure
    // set the signals for plotting
   po.setX(x.getv)
   po.setY(y1.getv)
   po.setColor(Color.GREEN)  // use GREEN color for plotting
```

```
po.setplotTitle2D(" Demonstrating 2-D plots")
po.setDottedLine()
po.mkplot()   // plot the first signal

po.setContinuousLine();   // set now to continous line plotting
po.setColor(Color.RED)
po.setlineWidth(15)   // set to thicker width
// redefine the new signals for plotting
hold("on");
po.setY(y2)   // we change only the Y signal

figure; po.mkplot()

var z = cos(4.5*x)
po.setZ(z)

po.setColor(Color.BLUE)  // change the plot color
po.setlineWidth(1)  // set the line width to 1
po.mkplot3D
```

Chapter 11 Applications with Numerical Recipes code

Model Fitting with Levenberg-Marquardt

The Levenberg-Marquardt method [12] fits a nonlinear model to the data. It works by varying smoothly between the extremes of the inverse-Hessian method and the steepest descent method. The inverse-Hessian method is used far from the minimum, switching continuously to the steepest descent as the minimum is approached. The Levenberg-Marquardt method works usually very well when plausible starting guesses for the parameters of the nonlinear model can be guested.

The example generates data that are the sum of two Gaussians (method *ag*), injects noise to them, and then tries to fit a Levenberg-Marquardt model to the synthetic data.

```
import com.nr.NRUtil.SQR
import com.nr.NRUtil.buildVector

import com.nr.model.FGauss
import com.nr.model.Fitmrq
import com.nr.ran.Normaldev

// Levenberg-Marquardt example

closeAll

  var  N = 200  // data array length
  var  MA=6   // the model's parameters
  var aa = Array(5.0, 1.0, 1.5, 2.0, 9.0, 3.0)
  var gguess = Array(4.5, 1.2, 2.8, 2.5, 8.9, 2.8)
  var  SPREAD = 0.01
  var  x = new Array[Double](N)
  var  y = new Array[Double](N)
  var  sig = new Array[Double](N)  // individual standard deviations
  var  a = aa.clone
```

```
var guess = gguess.clone

// Test Fitmrq
println("Testing Fitmrq")

var  ndev = new Normaldev(0.0, 1.0, 17)

var Np = 5000
var nd = new Vec(Np)  // test some normaldev points
for (k<-0 until Np)
 nd(k) = ndev.dev
 plot(nd)

figure(2); title("Normal  density")
var slicesX = 30
plot2d_histogram(nd.getv, slicesX, "Normal Probability Density")

// First try a sum of two Gaussians
// y(i) = a(0)*exp(-((x(i)-a(i))/a(2))^2+a(3)*exp(-((x(i)-a(i))/a(5))^2
// the actual unknown function
def ag(x: Double, a: Array[Double]) = {
   a(0)*exp(-SQR((x-a(1))/a(2)))+a(3)*exp(-SQR((x-a(4))/a(5)))
}

 var dx= 0.1
for (i <- 0 until N) {
       x(i) = 0.1*(i+1)    // x data values
     y(i) =  ag(dx*i, a)    // y data values
     y(i) *= (1.0+SPREAD*ndev.dev())   // add some noise to the y-values
    sig(i) = SPREAD*y(i)  // individual standard deviations
}

figure(3); plot(y,  Color.BLUE, "Sum of two Gaussians")
```

```
   var fgauss = new Fgauss()    // the user supplied function that calculates the
                                 // nonlinear fitting function and its derivatives
     // perform the Levenberg-Marquardt optimization
   var myfit = new Fitmrq(x, y, sig, guess, fgauss)
   myfit.fit()

// evaluate using the model coefficients
def evg(x: Double, mf: com.nr.model.Fitmrq) =
  ag(x, mf.a)

var yc = new Array[Double](y.length)
  // evaluate reconstructed Gaussian model
 for (i <- 0 until N)
   yc(i) = evg(dx*i, myfit)

figure(4); hold("on");    plot(yc, Color.GREEN, "Reconstructed" );
   plot(y,  Color.RED, "Original")

println( "chi-squared:  \n"+ myfit.chisq)
for (i <- 0 until MA)  println( myfit.a(i))
println
println("Uncertainties:")
for (i <- 0 until  MA)  println(sqrt(myfit.covar(i, i)))

println("Expected results:")

 for (i <- 0 until  MA)  println(a(i))



  var localflag = false  // signals an error in the estimated uncertainty
  var globalflag = false
 for (j<-0 until MA) {
       var j = 0
     localflag = abs(myfit.a(j)-a(j)) > 2.0*sqrt(myfit.covar(j)(j))  // a true value signals a bad fit
```

```
    globalflag = globalflag || localflag
    if (localflag) {
      println("*** Fitmrq: Fitted parameter "+j+ "  not within estimated uncertainty")

    }
  }
```

Minimization and Maximization of Functions

Powell method. Example 1

```
import com.nr.RealValueFun
import com.nr.min.Powell

// function to minimize with Powell method
// we define a function of two  variables
class funcToMinimize extends RealValueFun {
      def funk( x: Array[Double]) = {
            cos(x(0)*x(0))-sin(x(1)*x(1))- 8.7
        }
      }

 var  p = new Array[Double](2)


   // Test Powell
  var funcToMinimizeObj = new funcToMinimize()   //  create an instance of the function to
minimize
  var   pow1 = new Powell(funcToMinimizeObj)  // use the Powell minimizer for that particular
```

```
function
    p(0) = 2.0;  p(1) = 5.0;  //  a starting initial condition
    var p0 = pow1.minimize(p)  //  minimize that with Powell method
//   the function value obtained at the point of the computed minimum by the Powell's method
    var minPowell = funcToMinimizeObj.funk(p0)



   // define the  function being optimized, conveniently  for plotiing
   def f(x: Double, y: Double) = {
        var xa =  new Array[Double](2)
        xa(0) = x
        xa(1) = y
     funcToMinimizeObj.funk(xa)
   }


      // plot now the function in order the minima to be evident
close("all")
figure3d; fplot2d(f, 3.0, 3.5,  4.2, 4.8, Color.RED, true);
title("minimum at "+p0(0) +" , "+p0(1))
```

Powell method – Example 2

The example optimizes the function

$$f(x,y,z)=\frac{1}{2}-J_0\left((x-1)^2+(y-2)^2+(z-1)^2\right)$$

The starting point is P=(3/2, 3/2, 5/2) and the unit directions (1,0,0), (0,1,0), (0,0,1) are used as the initial directions.

```
/* optimize the function
        f(x, y, z) = 1/2 - J_0[(x-1)^2+(y-2)^2+(z-3)^2]
    with the Powell's method
   The script provides   powell with a starting point P of (3/2, 3//2, 5/2) and a set of initial directions
   here chosen to be the unit directions (1, 0, 0), (0, 1, 0) and (0, 0, 1)
```

```
  powell performs its one-dimensional minimizations with linmin */

// function to minimize with Powell method
// we define a function of two  variables

import com.nr.RealValueFun

class funcToMinimize extends RealValueFun {
      def funk( x: Array[Double]) = {
import com.nr.sf.Bessjy
val bj = new Bessjy // construct a Bessel object

  0.5-bj.j0( (x(0)-1.0)*(x(0)-1.0)+(x(1)-2.0)*(x(1)-2.0)+(x(2)-3.0)*(x(2)-3.0))
        }
      }


   // Test Powell
  var funcToMinimizeObj = new funcToMinimize()   //  create an instance of the function to
minimize
  import com.nr.min.Powell

  var   pow1 = new Powell(funcToMinimizeObj)  // use the Powell minimizer for that particular
function


 var NDIM = 3
// this matrix defines the initial set of directions
 var xi = eye(NDIM)

var p = new AD(3)

 var p0 = pow1.minimize(p, xi)  //  minimize that with Powell method
 println("iterations performed  = "+ pow1.iter)
```

Statistical Description of Data

Here, we present examples illustrating the statistical routines.

Testing avevar

The following example uses the *avevar()* method to compute the mean and variance of a set of numbers.

```
import com.nr.NRUtil.buildVector
import com.nr.stat.Moment.avevar
import com.nr.test.NRTestUtil.maxel
import com.nr.test.NRTestUtil.vecsub
import java.lang.Math.acos
import java.lang.Math.sin
import org.netlib.util.doubleW

 var  NPTS=10000  // number of points
 var  NBIN=101    // number of bins
 var  NPNB=NPTS+NBIN

 var  n=0
 var  pi=acos(-1.0)

   // Expected average, variance
 var  expect= Array(pi/2.0, 0.467401)

// since Java does not support passing of primitive types by reference, Numerical Recipes
// code wraps them in doubleW objects
 var ave=new doubleW(0)
 var vrnce=new doubleW(0)

 var temp=new Array[Double](NPNB)
 var e=buildVector(expect)
```

```
var  localflag=false; var globalflag=false;


  // Test avevar
  println("Testing avevar")

  // Create a sinusoidal distribution
  var x=0.0
  while (x<=pi) {
      var nlim=(0.5+sin(x)*pi/2.0*NPTS/NBIN).asInstanceOf[Int]
   for (k<-0 until  nlim) { temp(n) = x; n+=1 }
   x += pi/NBIN
   }

 var data = new   Array[Double](n)
 for (k<-0 until n) data(k) = temp(k)

// compute average and variance of data
  avevar(data, ave, vrnce)

  var observe  = Array(ave.`val`, vrnce.`val`)

  var o = buildVector(observe)

  var sbeps = 2.0e-4
  localflag = maxel(vecsub(e,o))>sbeps
  globalflag = globalflag || localflag
 if (localflag) {
    fail("*** moment: Reported average or variance of sinusoidal distribution was out of spec");
  }

  if (globalflag) println("Failed\n")
  else println("Passed\n")
```

Testing Student's t-Test for Significantly Different Means

```
// a test for Student's t-Test for significantly different means

import  com.nr.stat.Stattests.ttest
import org.netlib.util.doubleW
import com.nr.ran.Normaldev

 var NPTS=2000   // number of points
 var  data1 = new Array[Double](NPTS)
 var  data2 = new Array[Double](NPTS)
 var  data3 = new Array[Double](NPTS)
   // Test ttest
   println("Testing ttest")

   // Generate gaussian distributed data
 // normal distribution with mean=0.0, std=1.0, 17 is the initial seed for the random generator
    var  ndev1 = new Normaldev(0.0, 1.0, 17)
    var  ndev2 = new Normaldev(0.02, 2.0, 14)   // a distribution with close mean
   // Special case: identical distributions
      for (i<-0 until  NPTS)   data1(i) = ndev1.dev()
      for (i<-0 until  NPTS)  data2(i) =  ndev2.dev()

    var  twSimilar = new doubleW(0)
    var  pwSimilar = new doubleW(0)
    ttest(data1, data2, twSimilar, pwSimilar)
    var tvSimilar = twSimilar.`val`  // Student's t
    var pvSimilar = pwSimilar.`val`   // p-value

   println(" t value for similar = "+tvSimilar +", p value for similar = "+pvSimilar)
```

```
var  ndev3 = new Normaldev(2, 1.0, 14)   // a distribution with not close mean
for (i<-0 until  NPTS) data3(i) =  ndev3.dev()
var  twDifferent = new doubleW(0)
var  pwDifferent = new doubleW(0)
ttest(data1, data3, twDifferent, pwDifferent)

var tvDifferent = twDifferent.`val`
var pvDifferent = pwDifferent.`val`

println(" t value for different = "+tvDifferent + ", p value for different = "+pvDifferent)
```

Random Numbers (Chapter 7 NR)

Generating Logistic Deviates

```
var mmu = 2.0; var ssig=5.5
var ldobject = new com.nr.ran.Logisticdev(mmu, ssig, 33)

var N=60000  // number of points to evaluate the plot
var v = new Vec(N)

// generate the logistic deviates
for (k<-0 until N)
v(k) =  ldobject.dev()

figure(1);
plot(v, "Some logistic deviates")
  // plot them
var p = new ScalaSci.math.plot.Plot2DPanel("SOUTH");
```

```
   p.addHistogramPlot("Logistic deviates distribution", v, 600);
   new ScalaSci.math.plot.FrameView(p);
```

Normal Distribution

```
// the following script illustrates how to extent the normal distribution class of the
// Numerical Recipes with routines that compute the pdf, cdf, and inverse cdf

class NormalDist(m: Double, s: Double)  extends  com.nr.sf.Normaldist(m, s) {

  // evaluate the Probabilitry Density Function
  def evalPDF( low: Double, high: Double, npoints: Int = 2000) = {
      val  x = linspace(low, high, npoints)
      val  y = x map this.p  // map the PDF function from the Numerical Recipes implementation
(i.e. this.p)
      (x, y)
  }

  def evalCDF(low: Double, high: Double, npoints: Int = 2000) = {
      val x = linspace(low, high, npoints)
      val y = x map this.cdf // map the CDF function from the Numerical Recipes implementation
      (x, y)
  }

  def evalInvCDF(low: Double, high: Double, npoints: Int = 2000) = {
      val x = linspace(low, high, npoints)
      val y = x map this.invcdf // map the CDF function from the Numerical Recipes implementation
      (x, y)

  }
}


var m= 5.2; var s = 2.7
var nd  = new NormalDist(m, s)
```

```
var (axis, ndPDF) = nd.evalPDF(-5, 15, 3000)
figure(1); subplot(3,1,1); plot(axis, ndPDF, "Normal Distribution - Probability Density Function", Color.RED)

var (axis2, ndCDF) = nd.evalCDF(0, 15, 4000)
subplot(3,1,2); plot(axis2, ndCDF, "Normal Distribution - Cumulative Distribution Function", Color.BLUE)

var (axis3, invCDF) = nd.evalInvCDF(0.0001, 0.999, 3000)
subplot(3,1,3); plot(axis3, invCDF, "Normal Distribution - Inverse cumulative distribution function", Color.GREEN)
```

Normal Deviates by Transformation (Box-Muller)

```
var mmu = 2.0; var ssig=5.5
var seed = 22l   // an arbitrary seed
var ndobject = new com.nr.ran.Normaldev_BM(mmu, ssig, seed)

var N=60000  // number of points to evaluate the plot
var v = new Vec(N)

// generate the logistic deviates
for (k<-0 until N)
 v(k) =  ndobject.dev()

figure(1); plot(v, "Some normal deviates")
  // plot them
var p = new ScalaSci.math.plot.Plot2DPanel("SOUTH");
p.addHistogramPlot("normal deviates distribution", v, 600);
new ScalaSci.math.plot.FrameView(p);
```

Poisson deviates

```
var lambda = 5.0;
  var seed = 22l   // an arbitrary seed
  var  poissonObject = new com.nr.ran.Poissondev(lambda, seed)

 var N=60000  // number of points to evaluate the plot
 var v = new Vec(N)

 // generate the Poisson deviates
for (k<-0 until N)
 v(k) =  poissonObject.dev()

  figure(1); plot(v, "Some Poisson deviates")
    // plot them
  var p = new ScalaSci.math.plot.Plot2DPanel("SOUTH");
  p.addHistogramPlot("normal deviates distribution", v, 600);
  new ScalaSci.math.plot.FrameView(p);
```

Pearson Coefficient

```
import com.nr.NRUtil.buildVector
import com.nr.stat.Stattests.pearsn
import java.lang.Math.abs
import java.lang.Math.log
import org.netlib.util.doubleW

 var N=10
 var  doses = Array(56.1,64.1,70.0,66.6,82.0,91.3,90.0,99.7,115.3,110.0)
 var  spores = Array(0.11, 0.40, 0.37,0.48,0.75,0.66,0.71,1.20,1.01, 0.95)

  var   prob1=new doubleW(0)
  var   r1 = new doubleW(0)
  var   z1 = new doubleW(0)
  var   prob2=new doubleW(0)
```

```
var   r2 = new doubleW(0)
var   z2 = new doubleW(0)
var   sbeps=1.0e-16
var  expect = Array(0.9069586,0.2926505e-3,1.510110)
var dose = buildVector(doses)
var spore = buildVector(spores)
var dose2 = new Array[Double](N)
var  localflag = false
var  globalflag=false



// Test pearsn
println("Testing pearsn")

pearsn(dose, dose, r1, prob1, z1)
r1.`val`

localflag = (r1.`val`!= 1.0)
globalflag = globalflag || localflag
if (localflag) {
  fail("*** pearsn: Correlation of an array with itself is not reported as perfect")
}

localflag = (prob1.`val` > 1.0e-16)
globalflag = globalflag || localflag
if (localflag) {
  fail("*** pearsn: Probability for a perfect correlation was not zero")
}

for (i <- 0 until N)
   dose2(i) = 200.0-dose(i)

pearsn(dose, dose2, r2, prob2, z2)
```

```
localflag = (r2.`val` != -1.0)
globalflag = globalflag || localflag
if (localflag) {
  fail("*** pearsn: Correlation of an array with its negative is not reported as perfect")
}

localflag = (prob2.`val`> sbeps)
globalflag = globalflag || localflag
if (localflag) {
  fail("*** pearsn: Probability for a perfect anticorrelation was not zero")
 }

sbeps = 1.0e-6
pearsn(dose, spore, r1, prob1, z1)   // Data with known results
localflag = (abs(r1.`val` - expect(0)) > sbeps)
globalflag = globalflag || localflag
if (localflag) {
  fail("*** pearsn: Unexpected correlation coefficient for test data")
 }

localflag = (abs(prob1.`val` - expect(1))  >  sbeps)
globalflag = globalflag || localflag
if (localflag) {
  fail("*** pearsn: Unexpected probability for test data")
 }

localflag = (abs(z1.`val` - expect(2)) > sbeps)
globalflag = globalflag || localflag
if (localflag) {
  fail("*** pearsn: Unexpected Fisher's z coefficient for test data")
}

pearsn(spore, dose, r2, prob2, z2)
localflag = (r2.`val` != r1.`val`)
globalflag = globalflag || localflag
```

```
  if (localflag) {
    fail("*** pearsn: Correlation coefficient modified when arrays swapped");
  }

  localflag = (abs(prob2.`val`- prob1.`val`) > sbeps)
  globalflag = globalflag || localflag
  if (localflag) {
    fail("*** pearsn: Probability modified when arrays swapped")
  }

  localflag = (abs(z2.`val` - z1.`val`) > sbeps)
  globalflag = globalflag || localflag
  if (localflag) {
    fail("*** pearsn: Fisher's z modified when arrays swapped")
  }

  localflag = (abs(z2.`val` - 0.5*log((1+r2.`val`)/(1-r2.`val`)))) > sbeps
  globalflag = globalflag || localflag
  if (localflag) {
    fail("*** pearsn: Fisher's z not compatible with correlation coefficient")
  }

  if (globalflag)  println("Failed\n")
  else println("Passed\n");
```

Spearman Rank-Order Correlation Coefficient

```
import  com.nr.NRUtil.buildVector
import  com.nr.stat.Stattests.spear
import  java.lang.Math.abs

import org.netlib.util.doubleW
```

```
var  d1 = new doubleW(0)
var  zd1 = new doubleW(0)
var probd1 = new doubleW(0)
var  rs1 = new doubleW(0)
var probrs1 = new doubleW(0)
var  d2 = new doubleW(0)
var zd2 = new doubleW(0)
var probd2 = new doubleW(0)
var  rs2 = new doubleW(0)
var probrs2 = new doubleW(0)

var sbeps = 1.0e-6
var  adata = Array(0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0)
var  bdata = Array(9.0,8.0,7.0,6.0,5.0,4.0,3.0,2.0,1.0,0.0)
var  cdata = Array(1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,0.0) // Note 0.0 at end
var  edata = Array(1.0,2.0,1.0,2.0,1.0,2.0,1.0,2.0,1.0,2.0)
var  fdata = Array( 2.0,1.0,2.0,1.0,2.0,1.0,2.0,1.0,2.0,1.0)
// Expected results for each test case
var  ae = Array(0.0,-3.0,0.0026998,1.0,0.0)
var  be = Array(330.0,3.0,0.0026998,-1.0,0.0)
var  ce = Array(90.0,-1.363636,0.172682,0.454545,0.186905)
var  ee = Array(250.0,3.0,0.0026998,-1.0,0.0)
var   a = buildVector(adata)
var   b = buildVector(bdata)
var   c = buildVector(cdata)
var   e  = buildVector(edata)
var   f =  buildVector(fdata)
var  localflag = false
var globalflag = false


// Test spear
println("Testing spear")
```

```
// Test 1
spear(a, a, d1, zd1, probd1, rs1, probrs1)

localflag = localflag || (d1.`val` != ae(0))
globalflag = globalflag || localflag
if (localflag) {
  fail("*** spear: Squared difference of ranks not zero for identical distributions");
}

localflag = localflag || (zd1.`val` != ae(1))
globalflag = globalflag || localflag
if (localflag) {
  fail("*** spear: Unexpected standard deviation (should be -3) for special case")
}

localflag = localflag || abs(probd1.`val`-ae(2)) > sbeps
globalflag = globalflag || localflag
if (localflag) {
  fail("*** spear: Unexpected probability for d, given the standard deviation")
}

localflag = localflag || (rs1.`val` != ae(3))
globalflag = globalflag || localflag
if (localflag) {
  fail("*** spear: Spearman's rank correlation should be 1 for identical distributions")
}

localflag = localflag || (probrs1.`val`!= ae(4))
globalflag = globalflag || localflag
if (localflag) {
  fail("*** spear: Probrs should be zero for identical distributions")
}

// Test 2
spear(a, b, d1, zd1, probd1, rs1, probrs1)
```

```
//    System.out.printf(d1 << " %f\n", zd1 << " %f\n", probd1 << " %f\n", rs1 << " %f\n",
probrs1);
    localflag = localflag || (d1.`val` != be(0))
    globalflag = globalflag || localflag
    if (localflag) {
      fail("*** spear: Sum squared difference of ranks should be 2*(81+49+25+9+1)=330");
    }

    localflag = localflag || (zd1.`val` != be(1))
    globalflag = globalflag || localflag
    if (localflag) {
      fail("*** spear: Standard deviation should be (330/55)-3=3")
    }

    localflag = localflag || abs(probd1.`val`-be(2)) > sbeps
    globalflag = globalflag || localflag
    if (localflag) {
      fail("*** spear: Unexpected probability for d, given the standard deviation")
    }

    localflag = localflag || (rs1.`val` != be(3))
    globalflag = globalflag || localflag
    if (localflag) {
      fail("*** spear: Spearman's rank correlation should be -1 for perfect anticorrelation")
    }

    localflag = localflag || abs(probrs1.`val` - be(4))>1.0e-16
    globalflag = globalflag || localflag
    if (localflag) {
      fail("*** spear: Probrs should be zero for perfect anticorrelation")
    }

    // Test 3
    spear(b, a, d2, zd2, probd2, rs2, probrs2)
//    System.out.printf(d2 << " %f\n", zd2 << " %f\n", probd2 << " %f\n", rs2 << " %f\n",
probrs2);
```

```
localflag = localflag || (d1.`val` != d2.`val`) || (zd1.`val` != zd2.`val`) ||
   (probd1.`val` != probd2.`val`) || (rs1.`val` != rs2.`val`) || (probrs1.`val` != probrs2.`val`)
globalflag = globalflag || localflag;
if (localflag) {
  fail("*** spear: Results changed when two arrays were swapped (case 1)")
}

// Test 4
spear(a, c, d1, zd1, probd1, rs1, probrs1)

localflag = localflag || (d1.`val` != ce(0))
globalflag = globalflag || localflag
if (localflag) {
  fail("*** spear: Sum squared difference of ranks should be 9*(1^2)+1*(9^2)=90");
}

localflag = localflag || abs(zd1.`val`-ce(1)) > sbeps
globalflag = globalflag || localflag
if (localflag) {
  fail("*** spear: Standard deviation should be (90/55)-3=-1.363636")
  }

localflag = localflag || abs(probd1.`val`-ce(2)) > sbeps
globalflag = globalflag || localflag
if (localflag) {
  fail("*** spear: Unexpected probability for d, given the standard deviation")
}

localflag = localflag || abs(rs1.`val`-ce(3)) > sbeps
globalflag = globalflag || localflag
if (localflag) {
  fail("*** spear: Spearman's rank correlation should be 1-6*90/990=0.454545")
}

localflag = localflag || abs(probrs1.`val`-ce(4)) > sbeps
```

```
  globalflag = globalflag || localflag
  if (localflag) {
    fail("*** spear: Unexpected probability for rs, given the rank correlation")
  }

  // Test 5
  spear(c, a, d2, zd2, probd2, rs2, probrs2)
//   System.out.printf(d2 << " %f\n", zd2 << " %f\n", probd2 << " %f\n", rs2 << " %f\n",
probrs2);
  localflag = localflag || (d1.`val` != d2.`val`) || (zd1.`val` != zd2.`val`) ||
     (probd1.`val` != probd2.`val`) || (rs1.`val` != rs2.`val`) || (probrs1.`val` != probrs2.`val`)
  globalflag = globalflag || localflag;
  if (localflag) {
    fail("*** spear: Results changed when two arrays were swapped (case 2)")
   }

  // Test 6
  spear(e, f, d1, zd1, probd1, rs1, probrs1)
  localflag = localflag || (d1.`val` != ee(0))
  globalflag = globalflag || localflag
  if (localflag) {
    fail("*** spear: Sum squared difference of ranks should be 10*(8-3)^2=250")
  }

  localflag = localflag || (zd1.`val` != ee(1))
  globalflag = globalflag || localflag
  if (localflag) {
    fail("*** spear: Standard deviation should be (250-125)/55/(1-240/990)=3")
  }

  localflag = localflag || abs(probd1.`val`-ee(2)) > sbeps
  globalflag = globalflag || localflag
  if (localflag) {
    fail("*** spear: Unexpected probability for d, given the standard deviation")
  }
```

```
   localflag = localflag || (rs1.`val` != ee(3))
   globalflag = globalflag || localflag
   if (localflag) {
     fail("*** spear: Spearman's rank correlation should be -1 for perfect anticorrelation (case 2)")
   }

   localflag = localflag || (probrs1.`val` != ee(4))
   globalflag = globalflag || localflag
   if (localflag) {
     fail("*** spear: Probrs should be zero for perfect anticorrelation (case 2)")
   }

   // Test 7
   spear(f, e, d2, zd2, probd2, rs2, probrs2)
//    System.out.printf(d2 << " %f\n", zd2 << " %f\n", probd2 << " %f\n", rs2 << " %f\n",
probrs2);
   localflag = localflag || (d1.`val` != d2.`val`) || (zd1.`val` != zd2.`val`) ||
       (probd1.`val` != probd2.`val`) || (rs1.`val` != rs2.`val`) || (probrs1.`val` != probrs2.`val`)
   globalflag = globalflag || localflag
   if (localflag) {
     fail("*** spear: Results changed when two arrays were swapped (case 3)")
   }

   if (globalflag)  println("Failed\n")
   else  println("Passed\n");
```

F-test

```
import  com.nr.stat.Moment.avevar
import  com.nr.stat.Stattests.ftest
import  java.lang.Math.abs
import  java.lang.Math.sqrt

import com.nr.ran.Normaldev
```

```
var  NVAL = 11
var  NPTS = 10000
var  ave1 = new doubleW(0)
var  var1 = new doubleW(0)
var  f = new doubleW(0)
var  prob = new doubleW(0)
var  f1 = new doubleW(0)
var  f2 = new doubleW(0)

var  prob1 = new doubleW(0)
var  prob2 = new doubleW(0)

 var EPS = 0.01

 var fingerprint = Array(1.0,0.618852,0.32215,0.139461,0.0498994,
     0.0147196,0.00357951,0.000718669,0.000119432,1.64815e-5,  1.89557e-6)

 var  data1 = new Array[Double](NPTS)
 var  data2 = new Array[Double](NPTS)

 var  localflag = false
 var  globalflag = false

       // Test ftest
  println("Testing ftest")

   // Generate two gaussian distributions with different variances
   var  ndev=new Normaldev(0.0, 1.0, 17)

   for (j<-0  until  NPTS)   data1(j) = ndev.dev()
   figure(1);  plot(data1, Color.GREEN, "Gaussian"); title("First gaussian distribution")

   avevar(data1, ave1, var1)
```

```
for (j<-0 until  NPTS)  data1(j) -= ave1.`val`

for (j<-0 until  NPTS)  data2(j) = data1(j)

ftest(data1, data2, f, prob)

localflag = (f.`val` != 1.0)
globalflag = globalflag || localflag
if (localflag) {
  fail("*** ftest: ftest on identical distributions does not return f=1.0")
}

sbeps = 1.0e-10
localflag = (1.0-prob.`val` > sbeps)
globalflag = globalflag || localflag;
if (localflag) {
  fail("*** ftest: ftest on identical distributions does not return prob=1.0");
}

for (j<-0 until  NPTS) data2(j) = (1.0+EPS)*data1(j)
ftest(data1, data2, f1, prob1)
ftest(data2, data1, f2, prob2)

localflag = (f1.`val` != f2.`val`) || (prob1.`val` != prob2.`val`)
globalflag = globalflag || localflag
if (localflag) {
  fail("*** ftest: ftest not symmetrical with respect to data arrays");
}

var sbeps1 = 1.0e-14
var sbeps2 = 1.0e-6
for (i<-0 until  NVAL ) {
  var  factor = sqrt(1.0+i*EPS)
  for (j <- 0 until  NPTS)  data2(j) = factor * data1(j)
```

```
    ftest(data1, data2, f, prob)
    localflag = (f.`val`-1.0-i*EPS > sbeps1)
    globalflag = globalflag || localflag
    if (localflag) {
      fail("*** ftest: Variance ratio f is incorrect")
    }

//     System.out.printf(abs(prob-fingerprint[i]));
    localflag = (abs(prob.`val`-fingerprint(i)) > sbeps2)
    globalflag = globalflag || localflag
    if (localflag) {
      fail("*** ftest: Probabilities do not agree with previous fingerprint")
       }
  }

  if (globalflag)  println("Failed\n")
  else  println("Passed\n")
```

Classification and Inference (Chapter 16 NR)

Support Vector Machines

```
import com.nr.ci.Svm
import com.nr.ci.Svmgausskernel
import com.nr.ci.Svmlinkernel
import com.nr.ci.Svmpolykernel
import com.nr.ran.Normaldev
import com.nr.ran.Ran

  var  M = 1000; var N = 2
  var  omega = 1.3

  var   x = new Array[Double](2)
  var   y = new Array[Double](M)
  var  data = Array.ofDim[Double](M, N)
  var globalflag = false
  var localflag = false



  // Test Svm
  println("Testing Svm")

  // Create two disjoint sets of points
  var  myran=new Ran(17)
  for (i <-0 until M/2) {
   y(i) = 1.0
   var a = myran.doub()
   var b = 2.0*myran.doub()-1.0
   data(i)(0)=1.0+(a-b)
   data(i)(1)=1.0+(a+b)
  }
```

```
 for (i <- M/2 until M) {
   y(i) = -1.0;
   var a = myran.doub();
   var b = 2.0*myran.doub()-1.0
   data(i)(0) = -1.0-(a-b)
   data(i)(1) = -1.0-(a+b)
 }

 // Linear kernel
 var linkernel=new Svmlinkernel(data,y)
 var linsvm=new Svm(linkernel)
 var lambda=10
 var k = 0
 var breakVar = false
 while (breakVar == false)  {
   var test = linsvm.relax(lambda,omega)
   k += 1
  if (test < 1.0e-3 || k >  100) breakVar = true
 }

 var nerror = 0
 for (i<-0 until  M)  {
   var predictionError = ((y(i)==1.0) != (linsvm.predict(i) >= 0.0))
   nerror += (if  (predictionError) 1 else  0)
 }
println("Errors: "+ nerror)

 // Need to add tests for harder test case and resolve issue that the two
 // support vectors give an erroneous indication for two of the kernels above

 // Example similar to the book
var  ndev = new Normaldev(0.0, 0.5, 17)
 for (j<-0 until 4)  {   // Four quadrants
   for ( i <- 0 until (M/4) ) {
```

```
    k=  ((M/4)*j+i).asInstanceOf[Int]
    if (j == 0) {
      y(k) = 1.0
      data(k)(0) = 1.0 + ndev.dev()
      data(k)(1) = 1.0 + ndev.dev()
    } else if (j == 1) {
      y(k) = -1.0
      data(k)(0) = -1.0+ndev.dev()
      data(k)(1) = 1.0+ndev.dev()
    } else if (j == 2) {
      y(k) = 1.0
      data(k)(0) = -1.0 + ndev.dev()
      data(k)(1) = -1.0 + ndev.dev()
    } else {
      y(k) = -1.0
      data(k)(0) = 1.0+ndev.dev()
      data(k)(1) = -1.0+ndev.dev()
    }
   }
  }

  // Linear kernel
var   linkernel2 = new Svmlinkernel(data, y)
var  linsvm2 = new Svm(linkernel2)
println("Errors: ")
var xlambda = 0.001
while (xlambda < 10000) {
   k = 0
   var breakVar = false
  while (breakVar == false)  {
    var test = linsvm2.relax(lambda,omega)
    k += 1
    if (test < 1.0e-3 || k >  100) breakVar=true
   }
```

```
 nerror = 0
 for (i <- 0  until  M) {
var predictionError = ((y(i)==1.0) != (linsvm2.predict(i) >= 0.0))
 nerror += (if  (predictionError) 1 else  0)
 }
 println(nerror)

      // Test new data
 nerror=0
 for (j <-0 until 4) {   // Four quadrants
  for (i<- 0 until M/4) {
    if (j == 0) {
      var yy =1.0
      x(0) = 1.0+ndev.dev()
      x(1) = 1.0+ndev.dev()
    } else if (j == 1) {
      var yy= -1.0;
      x(0) = -1.0+ndev.dev()
      x(1) = 1.0+ndev.dev()
    } else if (j == 2) {
      var yy=1.0
      x(0) = -1.0+ndev.dev()
      x(1)= -1.0+ndev.dev()
    } else {
      var yy = -1.0
      x(0) = 1.0+ndev.dev()
      x(1)= -1.0+ndev.dev()
    }
 var predictionError = ((y(i)==1.0) != (linsvm2.predict(i) >= 0.0))
 nerror += (if  (predictionError) 1 else  0)
   }
 }

xlambda *= 10
println(nerror)
```

```
}
println()

// Polynomial kernel
var polykernel2 = new Svmpolykernel(data, y, 1.0, 1.0, 4.0)
var polysvm2=new Svm(polykernel2)
println("Errors: ")
xlambda = 0.001
while  (xlambda <  10000 ) {
 var k=0
 var breakVar = false
 while (breakVar==false) {
  var test = polysvm2.relax(lambda,omega)
  k += 1
  if (test < 1.0e-3 || k >  100) breakVar = true
 }
 // Test training set
 nerror = 0
 for (i <- 0 until M) {
 var predictionError = ((y(i)==1.0) != (linsvm2.predict(i) >= 0.0))
 nerror += (if  (predictionError) 1 else  0)
 }
 println(nerror)
 // Test new data
 nerror = 0
 for (j <- 0 until  4) {   // Four quadrants
  for (i <- 0 until M/4) {
   if (j == 0) {
    var yy=1.0;
    x(0) =1.0 + ndev.dev()
    x(1)=1.0 + ndev.dev()
   } else if (j == 1) {
    var yy = -1.0
    x(0) = -1.0 + ndev.dev()
```

```
      x(1) = 1.0 + ndev.dev()
    } else if (j == 2) {
      var yy=1.0
      x(0) = -1.0 + ndev.dev()
      x(1) = -1.0 + ndev.dev()
    } else {
      var yy = -1.0
      x(0) = 1.0+ndev.dev()
      x(1) = -1.0+ndev.dev()
    }

    var predictionError = ((y(i)==1.0) != (linsvm2.predict(i) >= 0.0))
    nerror += (if  (predictionError) 1 else  0)
   }
  }
  xlambda *= 10
  println(nerror)
}
println()

// Gaussian kernel
var gausskernel2 = new Svmgausskernel(data, y, 1.0)
var gausssvm2 = new Svm(gausskernel2)
println("Errors: ")
xlambda = 0.001
while (xlambda < 10000) {
  var k = 0
  var breakVar = false
  while (breakVar==false)  {
    var test = gausssvm2.relax( lambda, omega)
    k += 1
    if (test < 1.0e-3 || k >  100) breakVar = true
  }
  nerror = 0
  for (i<-0 until  M) {
```

```
    var predictionError = ((y(i)==1.0) != (linsvm2.predict(i) >= 0.0))
    nerror += (if  (predictionError) 1 else  0)
  }
  println(nerror)

  // Test new data
  nerror = 0
  for (j<-0 until 4) {   // Four quadrants
   for (i <-0 until  M/4) {
     if (j == 0) {
       var yy = 1.0
       x(0) = 1.0 + ndev.dev()
       x(1) = 1.0 + ndev.dev()
     } else if (j == 1) {
       var yy = -1.0
       x(0) = -1.0+ndev.dev()
       x(1) = 1.0+ndev.dev()
     } else if (j == 2) {
       var yy =1.0
       x(0) = -1.0+ndev.dev()
       x(1) = -1.0+ndev.dev()
     } else {
       var yy = -1.0
       x(0) = 1.0+ndev.dev()
       x(1) = -1.0+ndev.dev()
     }

    var predictionError = ((y(i)==1.0) != (linsvm2.predict(i) >= 0.0))
    nerror += (if  (predictionError) 1 else  0)
   }
  }
  xlambda *= 10
  println("%d    ",nerror)
}
println()
```

```
// Test the algorithm on test data after learning
// Do a scan over lambda to find best value

  localflag = false;
  globalflag = globalflag || localflag;
  if (localflag) {
    fail("*** Svm: **************************");

  }

  if (globalflag)  println("Failed\n")
  else   println("Passed\n")
```

Gaussian Mixture Models

```
import  com.nr.NRUtil.SQR
import  com.nr.NRUtil._
import  com.nr.test.NRTestUtil._
import  java.lang.Math.sqrt

import com.nr.ci.Gaumixmod
import com.nr.ran.Normaldev


  var NDIM=2;   var NMEANS=4; var NPT=1000
  var flag = 0.0;  var sqrt2=sqrt(2.0)
  var ffrac = Array(0.25,0.25,0.25,0.25)
  var mmeans = Array(0.0,0.0,0.75,0.0,-0.25,-0.25,0.33,0.66)
  var gguess = Array(0.1,0.1,0.7,0.1,-0.2,-0.3,0.3,0.5)
  var ssigma = Array(0.1,0.1,0.02,0.2,0.01,0.1,0.1,0.05)
  var vvec1 = Array(1.0,0.0,1.0,0.0,sqrt2,sqrt2,sqrt2,sqrt2)
```

```
  var vvec2 = Array(0.0,1.0,0.0,1.0,-sqrt2,sqrt2,-sqrt2,sqrt2)

  var frac = buildVector(ffrac)


  var offset = new Array[Double](NMEANS)
  var guess = buildMatrix(NMEANS, NDIM, gguess)
  var means = buildMatrix(NMEANS, NDIM, mmeans)
  var sigma = buildMatrix(NMEANS, NDIM, ssigma)
  var vec1 = buildMatrix(NMEANS, NDIM, vvec1)
  var vec2 = buildMatrix(NMEANS, NDIM, vvec2)
  var x = new RichDouble2DArray(NPT, NDIM)
  var globalflag = false

  // Test Gaumixmod
  println("Testing Gaumixmod")

  var    ndev = new Normaldev(0.0, 1.0, 17)

  // Generate four groups of data
  var k = 0
  for (i <- 0 until 4) {
    for (j <-0 until (NPT * frac(i)).asInstanceOf[Int]) {
      var d0 = sigma(i)(0) *  ndev.dev()
      var d1 = sigma(i)(1) *  ndev.dev()
      x(k, 0) = means(i)(0) + d0 * vec1(i)(0) + d1 * vec2(i)(0)
      x(k, 1) = means(i)(1) + d0 * vec1(i)(1) + d1 * vec2(i)(1)
      k+=1
    }
  }

  var xx =  x~    // transpose the array


var gmix = new Gaumixmod(x, guess)
```

```
var  i = 0
var  breakFlag = false
while ( i < 100 && breakFlag == false) {
  flag = gmix.estep()
  if (flag < 1.0e-6) breakFlag = true
  gmix.mstep()
  i += 1
 }

// check for convergence
//    System.out.println("  flag: %f\n", flag);
var localflag = (flag > 1.0e-6)
globalflag = (globalflag || localflag)
if (localflag) {
  fail("*** Gaumixmod: No solution with 100 iterations");
 }

// Check for correct determination of population fractions
//    System.out.printf(maxel(vecsub(gmix.frac,frac)));
var sbeps = 0.005
localflag = maxel(vecsub(gmix.frac,frac)) > sbeps
globalflag = (globalflag || localflag)
if (localflag) {
  fail("*** Gaumixmod: Population fractions not accurately determined");
  }

// Check for correct determination of means
for (i <- 0 until NMEANS)
   offset(i) = sqrt(SQR(gmix.means(i)(0) - means(i)(0))
  +SQR(gmix.means(i)(1)-means(i)(1)))
//    System.out.printf(maxel(offset));
localflag = maxel(offset) > 0.01
globalflag = (globalflag || localflag)
if (localflag) {
```

```
    fail("*** Gaumixmod: Means are incorrectly identified")

  }

  // Check for correct determination of covariance matrices
  for (i <- 0 until NMEANS) {
    println("%f %f\n", gmix.sig(i)(0)(0), gmix.sig(i)(0)(1))
    println("%f %f\n\n",gmix.sig(i)(1)(0), gmix.sig(i)(1)(1))
  }

  if (globalflag)  println("Failed\n")
  else  println("Passed\n")



var gmixTrans = (gmix.means)~

figure(1);
scatterPlotsOn; plot(xx)   // plot the four groups of data that we constructed for testing K-means
hold("on")
plotMarks( gmixTrans(0),  gmixTrans(1))
```

Integration of Ordinary Differential Equations

Lorenz Chaotic System Integration with Dormand-Prince fifth-order stepper

```
// evaluate the Lorenz chaotic attractor with Dormand-Prince fifth-order stepper
import  com.nr.test.NRTestUtil.maxel
import  com.nr.test.NRTestUtil.vecsub

import  com.nr.ode.DerivativeInf
import  com.nr.ode.Odeint
import  com.nr.ode.Output



 var ystart = Array( 0.1,  0.2,  0.12)    // starting values of integration
 var xx1 = 1.0; var xx2 = 10.0  // integrate from xx1 to xx2
 var atol = 0.001; var rtol = 0.01  // absolute tolerance and relative tolerance
 var h1 = 0.02  // guessed first stepsize
 var hminn = 0.01   // minimum allowed stepsize (can be zero)
 var nsavePoints = getInt("How many points to save ?", 5000)
 var outt = new Output(nsavePoints)
 var s = new com.nr.ode.StepperDopr5()



  // specifies the derivatives of the Lorenz system
 class LorenzDerivs extends AnyRef with  DerivativeInf {

  def derivs(x: Double, y: Array[Double], dydx: Array[Double]) = {
    lorenz_derivs(x, y, dydx)
  }
```

```
  def  lorenz_derivs(x: Double,   y: Array[Double],  dydx: Array[Double]) = {
     var xx = y(0); var yy = y(1); var zz = y(2)

    dydx(0) = 10*(yy-xx)
    dydx(1) = -xx*zz+143*xx-yy
    dydx(2) = xx*yy - 2.66667*zz

 }

  def  jacobian(x: Double, y: Array[Double], dfdx: Array[Double], dfdy: Array[Array[Double]])
= {}
 }


  var  derivss = new LorenzDerivs()

 var  lorenzODE = new  Odeint(ystart, xx1, xx2, atol, rtol,  h1,  hminn, outt, derivss,  s)
 lorenzODE.integrate()   // proceed with the integration

 // convert to RichDoubleDoubleArray in order to extract the variables for plotting
var yout = new RichDouble2DArray(outt.ysave)
var y0 = yout(0,::).getv
var y1 = yout(1,::).getv
var y2 = yout(2,::).getv
plot(y0, y1, y2,"Lorenz Attractor integrated with Dormand-Prince fifth-order stepper ")
```

Runge-Kutta

```
import  com.nr.test.NRTestUtil.maxel
import  com.nr.test.NRTestUtil.vecsub

import  com.nr.ode.DerivativeInf
import  com.nr.ode.Odeint
import  com.nr.sf.Bessjy

var  nvar = 4
val y = new Array[Double](nvar)
val dydx = new Array[Double](nvar)
val yout = new Array[Double](nvar)
val yexp = new Array[Double](nvar)

var globalflag=false

 // Test rk4
println("testting  rk4")


   var x = 1.0
   var h = 0.1

   var bess = new Bessjy()
   for (i <- 0 until nvar) {
     y(i) = bess.jn(i, x)
     yexp(i) = bess.jn(i, x+h)
   }

   var  rk4_derivs = new RK4_derivs()

   rk4_derivs.rk4_derivs(x, y, dydx)
```

```
  Odeint.rk4(y, dydx, x, h, yout, rk4_derivs)

  for (i <- 0 until  nvar)
    println( yout(i), yexp(i))

  var sbeps = 1.0e-6
  println(maxel(vecsub(yout,yexp)))
  var localflag = maxel(vecsub(yout,yexp)) > sbeps
  globalflag = globalflag || localflag
  if (localflag) {
    fail("*** rk4: Inaccurate Runge-Kutta step")
  }

  if (globalflag)  println("Failed\n")
  else  println("Passed\n")


class RK4_derivs extends AnyRef with  DerivativeInf {

  def derivs(x: Double, y: Array[Double], dydx: Array[Double]) = {
    rk4_derivs(x, y, dydx)
  }

  def  rk4_derivs(x: Double,   y: Array[Double],  dydx: Array[Double]) = {
    dydx(0) = -y(1)
    dydx(1) = y(0)- (1.0 / x) * y(1)
    dydx(2) = y(1) - (2.0 / x) * y(2)
    dydx(3) = y(2) - (3.0 / x) * y(3)
  }

  def  jacobian(x: Double, y: Array[Double], dfdx: Array[Double], dfdy: Array[Array[Double]])
= {}
  }
```

Bulirsch-Stoer Method

```
import  com.nr.test.NRTestUtil.maxel
import  com.nr.test.NRTestUtil.vecsub

import com.nr.ode._
import com.nr.sf.Bessjy


 class rhs_StepperBS  extends AnyRef with DerivativeInf  {
    def   derivs(x: Double,   y: Array[Double],  dydx: Array[Double]) = {
     dydx(0) = -y(1)
     dydx(1) = y(0) - (1.0 / x) * y(1)
     dydx(2) = y(1) - (2.0 / x) * y(2)
     dydx(3) = y(2) - (3.0 / x) * y(3)
   }
   def  jacobian(x: Double,  y: Array[Double],  dfdx: Array[Double],  dfdy: Array[Array[Double]])
= {}
 }


   var nvar = 4
   var atol = 1.0e-6;     var  rtol = atol;   var  h1 = 0.01;   var hmin = 0.0;   var  x1 = 1.0;   var  x2
= 2.0;


   var  y = new Array[Double](nvar)
   var  yout = new Array[Double](nvar)
   var  yexp = new Array[Double](nvar)



   // Test StepperBS
   println("Testing StepperBS")


   var bess = new Bessjy()
```

```
for (i<-0  until  nvar) {
  y(i) = bess.jn(i, x1)
  yexp(i) = bess.jn(i, x2)
 }

var  out = new Output(1500)
var  d = new rhs_StepperBS()
var  s = new StepperBS()
var  ode = new Odeint(y, x1, x2, atol, rtol, h1, hmin, out, d, s)
 ode.integrate()

 for (i <- 0  until  nvar)  {
  yout(i) = out.ysave(i)(out.count-1)
  println(yout(i)+"  "+ yexp(i))
 }

 var  sbeps = 1.0e-8
 var  integrationError = maxel(vecsub(yout, yexp))

 if (integrationError < sbeps)
  println("Bulirsch-Stoer Method success")
 else
  println("Bulirsch-Stoer Method failure")
```

Loading and Saving from MATLAB's .mat files

ScalaLab provides some support for loading/saving MATLAB .mat files. Specifically *scalar* values and *arrays* can be handled. Class *ScalaSci.math.io.MatIO* implements some MATLAB .mat compatibility functionality. Specifically, the routines provided by this class are:

```
 // writes to the MATLAB .mat file the contents of the variable variableNameToSave of type double [] of the ScalaSci workspace
 public static  boolean  save(String fileName, double []  varValues, String varName)

// writes to the MATLAB .mat file the contents of the variable variableNameToSave of type double [][] of the ScalaSci workspace
 public static  boolean  save(String fileName, double [][]  varValues, String varName)

// writes to the MATLAB .mat file the contents of the variable varName of type Vec  of the ScalaSci workspace
  public static  boolean  save(String fileName, ScalaSci.Vec  vecValues, String varName)

 // save a Mat
public static  boolean  save(String fileName, ScalaSci.Mat  matValue, String varName)

// save a Matrix
public static  boolean  save(String fileName, ScalaSci.Matrix  matValue, String varName)

/ save an EJML.Mat
public static  boolean  save(String fileName, ScalaSci.EJML.Mat  matValue, String varName)


// save an MTJ.Mat
public static  boolean  save(String fileName, ScalaSci.MTJ.Mat  matValue, String varName)


// save an CommonMaths.Mat
public static  boolean  save(String fileName, ScalaSci.CommonMaths.Mat  matValue, String varName)

// save a JBLAS .Mat
```

```
public static  boolean  save(String fileName, ScalaSci.JBLAS.Mat  matValue, String varName)

// save a RichDoubleDoubleArray
public static  boolean  save(String fileName, ScalaSci.RichDoubleDoubleArray  matValue, String varName)

// loads the MATLAB .mat file contents to ScalaSci workspace
public static  int  load(String fileName)
```

When loading the contents of MATLAB .mat files two hashtables are very useful for keeping the names of the MATLAB variables and the corresponding data. These variables are:

```
static public Hashtable<String, Double> scalarValuesFromMATLAB = new Hashtable<String, Double>();

static public Hashtable<String, double[][]> arrayValuesFromMATLAB = new Hashtable<String, double[][]>();
```

The *load()* routines use these hashtables in order to permit the recovery of the MATLAB variables and their accompanied values.

In order to demonstrate the basic relevant routines by examples, we provide one .zip file at the Downloads section that has zipped some MATLAB mat files, obtained from the programming exercises accompanying the excellent Simon Haykin's "*Neural Networks and Learning Machines*" book. Therefore, you can download and run the following examples.

The following code loads the .mat files:

```
load("gridseq.mat")
load("c.mat")
load("C.mat")
load("nCor_ekf.mat")
load("seq.mat")
```

Then we can obtain a list of the names of the variables loaded from the .mat files using:

```
matVars
```

The above command displays separately the names of the *scalars* and the *matrices* that are read from the .mat files. Using these names, and remembering that the values are kept with hash tables, it is easy for the *getMatArray()* routine to recover the arrays:

```
val data_shuffled_ini = getMatArray("data_shuffled_ini")
val seq = getMatArray("seq")
```

```
val nCor_ekf = getMatArray("nCor_ekf")
val gridseq = getMatArray("gridseq")
val C = getMatArray("C")
```

The *whos* MATLAB like command, can be used to display the user defined variables and their types at the ScalaLab workspace, i.e.

```
whos
```

Also, the *whosv* command, displays in addition and the values of the variables, i.e.

```
whosv
```

Band Matrices

The band matrix type in ScalaLab is build upon the MTJ band matrix type. The later exploits the JLAPACK routines that operate effectively on band matrices.

We construct a band matrix by defining the size of its main diagonal *N*, and the sizes of the *lower* and *upper* subbands, *kl* and *ku*. For example:

val N = 8 // size of the main diagonal

val kl = 2 // lower diagonal

val ku = 3 // upper diagonal

val bm = BandMat(N, kl, ku)

We can construct another band matrix consisting of ones of the same size as:

val bm1 = ones(N, kl, ku)

And some other similar useful routines:

val bmr = rand(N, kl, ku) // a random band matrix

val bmz = zeros(N, kl, ku) // a zero band matrix

val bmf = fill(N, kl, ku, 2.2) // a band matrix filled with the passed value (i.e. 2.2)

We can perform the usual mathematical functions directly on band matrices, e.g. :

val bmsine = sin(bmr)

val bmexp = exp(bmr)

val bmlog = log(bmr)

val bmpow4 = pow(bmr, 4)

val bmceil = ceil(bmr)

We can have expressions using band matrices, e.g.

*val bmexpr = 4.5*bmr+bmr - bmr*8*

Sparse Matrices

The Sparse Matrix support with the **Sparse** Class. ScalaLab integrates support for sparse matrices based on the CSparse implementation of Timothy A. Davis, translated to Java by Piotr Wendykier. We describe here some sparse matrix operations by means of examples.

Handling sparse matrices

Suppose that we have a sparse matrix that is stored in a file in triplet format, e.g. the first entry:

2 2 3.0, means *a(2,2) = 3.0*. The file that stores the matrix suppose that it has the following contents:

2 2 3.0

1 0 3.1

3 3 1.0

0 2 3.2

1 1 2.9

3 0 3.5

3 1 0.4

1 3 0.9

0 0 4.5

2 1 1.7

Suppose that we store the matrix in a file: */home/sp/NBProjects/csparseJ/CSparseJ/matrix/t1*

The command to load the sparse matrix is:

var s = loadSparse("/home/sp/NBProjects/csparseJ/CSparseJ/matrix/t1")

We can display its contents with:

s.print

s.display

The former prints the contents in the format close to the internal representation while the later in a two-dimensional array format.

We can access an element of the sparse matrix as usual, e.g.

val s22 = s(2,2)

takes the corresponding element of the matrix.

We can also assign new values as usual, e.g.

s(1,2) = 12

We can add and multiply two sparse matrices as usual `

val s2 = s+s

*val sMs = s*s*

The transpose of a sparse matrix is obtained with the operator: ~

var ts = s~

We can add sparse matrices as usual, e.g.

var s2 = s+s

Also, multiplication is as a matrix multiplication, e.g.

*var sm = s*s*

We can transform a sparse matrix to an Array[Array[Double]] as

var da = toDouble(s)

Conversely we can transform an Array[Array[Double]] to a Sparse matrix, e.g.

var dd = Array.ofDim[Double](3,4)

dd(2)(3) = 23

dd(1)(2) = 12

var sparseDD = fromDoubleArray(dd)

We can add, subtract and multiply Sparse matrices with numbers, e.g.

var x10 = s+10

var x10ic = 10+s // using implicit conversion

*var y10 = s*10*

*var y10ic = 10*s // using implicit conversion*

We can also negate the matrix, e.g.

var sm = -s

Applying a function to all the matrix elements with map can be very useful, e.g. :

*def f(x: Double) = x*x*1000   // some function*

var sf = s map f // apply the function

MTJ Sparse Matrices

MTJ offers many sparse matrix classes and associated iterative algorithms for the solution of classical problems.

ScalaLab started to implement a higher-level interface to this functionality. For example, the CCMatrix class implements a Sparse class based on the Compressed Column Storage format of MTJ.

Example of Compressed Column Storage format

Here is an example illustrating some basic operations.

import no.uib.cipr.matrix

import no.uib.cipr.matrix.sparse

var nrows = 10; var ncols = 10

var d = Array.ofDim[Double](nrows, ncols)

d(2)(3) = 10

d(4)(4) = 44

```
// create a sparse matrix from the double [][] array
var sd = new CCMatrix(d)

var mtjCCM = sd.getCCM()   // get the MTJ based column cmpressed matrix representation

d(3)(5) =35
var sd2 = new CCMatrix(d)

// set an entry using putAt
sd(2, 1) =  21

// get entries using implicitly getAt
var elem2_1 = sd(2,1)
var elem2_2 = sd(2,2)



// test matrix addition
var sd1  = sd+sd2
sd
sd1

var sd10 = sd*100
sd10

var sdd = sd1-sd1
```

We can solve also a sparse linear system as:

```
var a = AAD("1, 2, 0; -1, 0, -2;  -3, -5, 1")

var b = Array(3.0, -5, -4)
var am = new CCMatrix(a)

var sol = ScalaSci.CCMatrix.BiCGSolve(am, b)
```

*var residual = max(a*sol-b) // should be very small*

Example on solving sparse systems

```
var filename = "/home/sp/NBProjects/csparseJ/CSparseJ/matrix/t1"

// load the sparse matrix stored in triplet format
var A = loadSparse(filename)

var b = vrand(A.Nrows).getv()

var x = solve(A, b)  // solve the system with the CSparse method

var Ad = SparseToDoubleArray(A)  // convert to double array

var residual = Ad*x - b  // verify: should be near zero


  // convert to an MTJ CCMatrix
 var ccms =    CSparseToCCMatrix(A)

 var xmtj = solve(ccms, b) // solve with the MTJ based iterative solver

var df = xmtj-x  // verify that the two solutions are equal
```

Apache Common Maths

The Apache Common Math is a powerful, elegant library for Java. It is included in the default ScalaLab installation. Here material from the Apache Common Maths user guide is adapted for ScalaLab. The Java code examples of the user guide are implemented in ScalaSci that is both scriptable and higher-level. In essence, we adapt the Java examples, to Scala code executed conveniently within the scripting environment of ScalaLab. Since Apache Common Math is a very useful library, ScalaLab performs by default the relevant imports to facilitate the user.

In order to use the Apache common maths library the relevant imports are performed implicitly. Therefore the ScalaLab programmer can refer conveniently to the routines of the Apache common maths library with their short names.

1. Statistics

Overview

The statistics package provides frameworks and implementations for basic Descriptive statistics, frequency distributions, bivariate regression, and t-, chi-square and ANOVA test statistics.

Descriptive Statistics

The *stat* package includes a framework and default implementations for the following Descriptive statistics:

- arithmetic and geometric means
- variance and standard deviation
- sum, product, log sum, sum of squared values
- minimum, maximum, median, and percentiles
- skewness and kurtosis
- first, second, third and fourth moments

With the exception of percentiles and the median, all of these statistics can be computed without maintaining the full list of input data values in memory. The stat package provides interfaces and implementations that do not require value storage as well as implementations that operate on arrays of stored values.

The top level interface is *UnivariateStatistic*. This interface, implemented by all statistics, consists of *evaluate()* methods that take *Array[Double]* as arguments and return the value of the statistic. This interface is extended by *StorelessUnivariateStatistic* , which adds *increment(), getResult()* and associated methods to support "storageless" implementations that maintain counters, sums or other state information as values are added using the *increment()* method.

Abstract implementations of the top level interfaces are provided in *AbstractUnivariateStatistic* and *AbstractStorelessUnivariateStatistic* respectively.

Each statistic is implemented as a separate class, in one of the subpackages (moment, rank, summary) and each extends one of the abstract classes above (depending on whether or not value storage is required to compute the statistic). There are several ways to instantiate and use statistics. Statistics can be instantiated and used directly, but it is generally more convenient (and efficient) to

access them using the provided aggregates, *DescriptiveStatistics* and *SummaryStatistics.*

DescriptiveStatistics maintains the input data in memory and has the capability of producing "rolling" statistics computed from a "window" consisting of the most recently added values.

SummaryStatistics does not store the input data values in memory, so the statistics included in this aggregate are limited to those that can be computed in one pass through the data without access to the full array of values.

Aggregate	Statistics Included	Values stored?	"Rolling" capability?
DescriptiveStatistics	min, max, mean, geometric mean, sum, sum of squares, standard deviation, variance, percentiles, skewness, kurtosis, median	Yes	Yes
SummaryStatistics	min, max, mean, geometric mean, sum, sum of squares, standard deviation, variance	No	No

SummaryStatistics can be aggregated using *AggregagateSummaryStatistics.* This class can be used to concurrently gather statistics for multiple datasets as well as for a combined sample including all of the data.

Neither *DescriptiveStatistics* nor *SummaryStatistics* is thread-safe.

SynchronizedDescriptiveStatistics and *SynchronizedSummaryStatistics*, respectively, provide thread-safe versions for applications that require concurrent access to statistical aggregates by multiple threads. *SynchronizedMultivariateSummaryStatistics* provides thread-safe *MultivariateSummaryStatistics.* There is also a utility class, *StatUtils*, that provides static methods for computing statistics directly from *Array[Double]*.

Here are some examples showing how to compute Descriptive statistics.

Using the *DescriptiveStatistics* aggregate (values are stored in memory):

```
// Get a DescriptiveStatistics instance
var stats = new DescriptiveStatistics
var inputArray = Array(0.2, 0.3444, -4.556, 0.344, 2.345)
// add the data from the array
for (i<-0 until inputArray.length)
 stats.addValue(inputArray(i))

 // Compute some statistics
var mean = stats.getMean
var std = stats.getStandardDeviation
var kurtosis = stats.getKurtosis
```

Using the *SummaryStatistics* aggregate (values are not stored in memory):

```
// Get a SummaryStatistics instance
var stats = new SummaryStatistics
// read data from a List of values
// adding values and updating sums, counters, etc.
var listOfValues = List(0.333, -2.34, 4.333, 0.3333, -0.33322)
// add the data from the list
listOfValues foreach ( stats.addValue(_) )

 // Compute some statistics
var mean = stats.getMean
var std = stats.getStandardDeviation
```

Using the *StatUtils* utility class:

```
var arrayOfValues = Array(0.333, -2.34, 4.333, 0.3333, -0.33322)
var mean = StatUtils.mean(arrayOfValues)
var std = StatUtils.variance(arrayOfValues)
var median = StatUtils.percentile(arrayOfValues, 50.0)
// compute the mean of the first three values in the array
var mn = StatUtils.mean(arrayOfValues, 0, 3)
```

Compute statistics in a thread-safe manner

Use a *SynchronizedDescriptiveStatistics* instance

```
// Create a SynchronizedDescriptiveStatistics instance and
// use it as any other DescriptiveStatistics instance
var stats: DescriptiveStatistics = new SynchronizedDescriptiveStatistics()
```

Frequency distributions

Frequency provides a simple interface for maintaining counts and percentages of discrete values.

Strings, integers, longs and chars are all supported as value types, as well as instances of any class that implements *Comparable*. The ordering of values used in computing cumulative frequencies is by default the *natural ordering*, but this can be overriden by supplying a *Comparator* to the constructor. Adding values that are not comparable to those that have already been added results in an *IllegalArgumentException.*

Here are some examples.

Compute a frequency distribution based on integer values.

```
var f = new Frequency
f.addValue(1)
f.addValue(new Integer(1))
f.addValue(2)
f.addValue(new Integer(-1))
```

```
println( f.getCount(1))
println( f.getCumPct(0))
println(f.getPct(new Integer(1)))
println(f.getCumPct(-2))
println(f.getCumPct(10))
```

Count string frequencies

Using case-sensitive comparison, alpha sort order (natural comparator):

```
var f = new Frequency
f.addValue("one")
f.addValue("One")
f.addValue("oNe")
f.addValue("Z")
println(f.getCount("one"))
println(f.getCumPct("Z"))
println(f.getCumPct("Ot"))
```

Using case-insensitive comparator:

```
var f = new Frequency(String.CASE_INSENSITIVE_ORDER)
f.addValue("one")
f.addValue("One")
f.addValue("oNe")
f.addValue("Z")
println(f.getCount("one"))
println(f.getCumPct("Z"))
println(f.getCumPct("Ot"))
```

Simple regression

SimpleRegression provides ordinary least squares regression with one independent variable estimating the linear model:

*y = intercept + slope * x*

Standard errors for intercept and slope are available as well as ANOVA, r-square and Pearson's r statistics.

Observations (x,y pairs) can be added to the model one at a time or they can be provided in a 2-dimensional array. The observations are not stored in memory, so there is no limit to the number of observations that can be added to the model.

Usage Notes:

- When there are fewer than two observations in the model, or when there is no variation in the x values (i.e. all x values are the same) all statistics return NaN. At least two observations with different x coordinates are required to estimate a bivariate regression model.
- getters for the statistics always compute values based on the current set of observations -- i.e., you can get statistics, then add more data and get updated statistics without using a new instance. There is no "compute" method that updates all statistics. Each of the getters performs the necessary computations to return the requested statistic.

Implementation Notes:

- As observations are added to the model, the sum of x values, y values, cross products (x times y), and squared deviations of x and y from their respective means are updated using updating formulas defined in "Algorithms for Computing the Sample Variance: Analysis and Recommendations", Chan, T.F., Golub, G.H., and LeVeque, R.J. 1983, American Statistician, vol. 37, pp. 242-247, referenced in Weisberg, S. "Applied Linear Regression". 2nd Ed. 1985. All regression statistics are computed from these sums.
- Inference statistics (confidence intervals, parameter significance levels) are based on the assumption that the observations included in the model are drawn from a *Bivariate Normal Distribution*

Here are some examples.

```
var regression = new SimpleRegression()
regression.addData(1d, 2d)
// At this point, with only one observation, all regression statistics will return NaN
regression.addData(3d, 3d)
// with only two observations, slope and intercept can be computed
// but inference statistics will return NaN
regression.addData(3d, 3d)
// Compute some statistics based on observations added so far
// displays the intercept of regression line
println(regression.getIntercept())
// displays slope of regression line
println(regression.getSlope())
// displays slope standard error
println(regression.getSlopeStdErr())
```

Use the regression model to predict the y value for a new x value

```
// displays predicted y value for x = 1.5
println(regression.predict(1.5d))
```

More data points can be added and subsequent getXxx calls will incorporate the additional data in statistics.

The following example estimates a model from a double[][] array of data points

```
// Instantiate a regression object and load a dataset

var data = Array(Array(1.0, 3.0), Array(2.0, 5.0), Array(3, 7.0), Array(4.0, 14), Array(5.0, 11.0))

var regression = new SimpleRegression()
regression.addData(data)  // add thye data as a two-dimensional vector
// estimate regression model based on data
// displays the intercept of regression line
println(regression.getIntercept())
// displays slope of regression line
println(regression.getSlope())
// displays slope standard error
println(regression.getSlopeStdErr())
```

More data points (even another double[][] array) can be added and subsequent getXxx calls will incorporate the additional data in statistics.

Multiple linear regression

The classes *OLSMultipleLinearRegression* and *GLSMultipleLinearRegression* provide least squares regression to fit the linear model:

Y=X*b+u

where Y is an n-vector *regressand*, X is a [n,k] matrix whose k columns are called *regressors*, b is k-vector of *regression parameters* and u is an n-vector of error terms or residuals.

OLSMultipleLinearRegression provides Ordinary Least Squares (OLS) Regression, and *GLSMultipleLinearRegression* implements Generalized Least Squares (GLS).

Data for OLS models can be loaded in a single double[] array, consisting of concatenated rows of data, each containing the regressand (Y) value, followed by regressor values; or using a double[][] array with rows corresponding to observations. GLS models also require a double[][] array representing the covariance matrix of the error terms.

Usage Notes:

- Data are validated when invoking any of the *newSample*, *newX*, *newY* or *newCovariance* methods and IllegalArgumentException is thrown when input data arrays do not have matching dimensions or do not contain sufficient data to estimate the model.
- By default, regression models are estimated with intercept terms. In the notation above, this implies that the X matrix contains an initial row identically equal to 1. X data supplied to the newX or newSample methods should not include this column - the data loading methods will create it automatically. To estimate a model without an intercept term, set the noIntercept property to true.

Here are some examples.

OLS regression

```
var regression = new OLSMultipleLinearRegression()
var y = Array(11.0, 12.0, 13.0, 14.0, 15.0, 16.0)
var x = new Array[Array[Double]](6)
x(0) = Array(0.0, 0.0, 0.0, 0.0, 0.0)
x(1) = Array(2.0, 0, 0, 0, 0)
x(2) = Array(0, 3.0, 0, 0, 0)
x(3) = Array(0, 0, 4.0, 0, 0)
x(4) = Array(0, 0, 0, 5.0, 0)
x(5) = Array(0, 0, 0, 0, 6.0)
regression.newSampleData(y, x)
// get regression parameters and diagnostics
var beta = regression.estimateRegressionParameters()
var residuals = regression.estimateResiduals()
var parametersVariance = regression.estimateRegressionParametersVariance()
var regressandVariance = regression.estimateRegressandVariance()
var rSsquared = regression.calculateRSquared()
var sigma = regression.estimateRegressionStandardError()
```

GLS regression

Instantiate a GLS regression object and load a dataset:

var regression = new GLSMultipleLinearRegression()
var y = Array(11.0, 12.0, 13.0, 14.0, 15.0, 16.0)
var x = new Array[Array[Double]](6)
x(0) = Array(0.0, 0, 0, 0, 0)
x(1) = Array(2.0, 0, 0, 0, 0)
x(2) = Array(0, 3.0, 0, 0, 0)
x(3) = Array(0, 0, 4.0, 0, 0)
x(4) = Array(0, 0, 0, 5.0, 0)
x(5) = Array(0, 0, 0, 0, 6.0)
var omega = new Array[Array[Double]](6)
omega(0) = Array(1.1, 0, 0, 0, 0, 0)
omega(1) = Array(0, 2.2, 0, 0, 0, 0)
omega(2) = Array(0, 0, 3.0, 0, 0, 0)
omega(3) = Array(0, 0, 0, 4.4, 0, 0)
omega(4) = Array(0, 0, 0, 0, 5.5, 0)
omega(5) = Array(0, 0, 0, 0, 0, 6.6)
regression.newSampleData(y, x, omega)

Rank transformations

Some statistical algorithms require that input data be replaced by ranks. The *org.apache.commons.math.stat.ranking* package provides rank transformation. *RankingAlgorithm* defines the interface for ranking. *NaturalRanking* provides an implementation that has two configuration options.

- *Ties strategy* determines how ties in the source data are handled by the ranking
- *NaN strategy* determines how NaN values in the source data are handled

Examples:

var ranking = new NaturalRanking(NaNStrategy.MINIMAL, TiesStrategy.MAXIMUM)
var data = Array(20, 17, 30, 42.3, 17, 50, Double.NaN, java.lang.Double.NEGATIVE_INFINITY, 17)
var ranks = ranking.rank(data)

results in ranks containing Array(6.0, 5.0, 7.0, 8.0, 5.0, 9.0, 2.0, 2.0, 5.0),

new NaturalRanking(NaNStrategy.REMOVED, TiesStrategy.SEQUENTIAL).rank(data)

returns Array(5.0, 2.0, 6.0, 7.0, 3.0, 8.0, 1.0, 4.0)

The default NaNStrategy is NaNStrategy.MAXIMAL. This makes NaN values larger than any other value (including Double.POSITIVE_INFINITY). The default *TiesStrategy* is *TiesStrategy.AVERAGE* , which assigns tied values the average of the ranks applicable to the sequence of ties.

Covariance and correlation

The *org.apache.commons.math.stat.correlation* package computes covariances and correlations for pairs of arrays or columns of a matrix. *Covariance* computes covariances, *PearsonsCorrelation* provides Person's Product-Moment correlation coefficients and *SpearmansCorrelation* computes Spearman's rank correlation.

Implementation Notes

- Unbiased covariances are given by the formula

$$cov(X,Y)=\frac{\sum_i (x_i-E(X))(y_i-E(Y))}{n-1}$$

where E(X) is the mean of X and E(Y) is the mean of Y values. Non-bias-corrected estimates use n in place of n-1. Whether or not covariances are bias-corrected is determined by the optional parameter, "biasCorrected,", which defaults to true.

- *PearsonsCorrelation* computes correlations defined by the formula

$$cor(X,Y)=\frac{\sum_i (x_i-E(X))(y_i-E(Y))}{(n-1)s(X)s(Y)}$$

where E(X) and E(Y) are means of X and Y and s(X), s(Y) are standard deviations.

- *SpearmansCorrelation* applies a rank transformation to the input data and computes Pearson's correlation on the ranked data. The ranking algorithm is configurable. By default, *NaturalRanking* with default strategies for handling ties and NaN values is used.

Examples:

Covariance of 2 arrays

To compute the unbiased covariance between 2 double arrays, x and y, use:

```
var N = 40
var x = vrand(N).getv  // a random Array[Double]
var y = vrand(N).getv  // a second one
// unbiased covariance
var unbiasedCov = new Covariance().covariance(x, y)
// non-biased corrected covariance
var nonBiasedCorrectedCov = new Covariance().covariance(x, y, false)
```

Covariance Matrix

Data Generation

The Commons Math random package includes utilities for

- generating random numbers
- generating random vectors
- generating random strings
- generating cryptographically secure sequences of random numbers or strings
- generating random samples and permutations
- analyzing distributions of values in an input file and generating values "like" the values in the file
- generating data for grouped frequency distributions or histograms

The source of random data used by the data generation utilities is pluggable. By default, the JDK-supplied PseudoRandom Number Generator (PRNG) is used, but alternative generators can be "plugged in" using an adaptor framework, which provides a generic facility for replacing java.util.Random with an alternative PRNG. Other very good PRNG suitable for Monte-Carlo analysis (but not for cryptography) provided by the library are the Mersenne twister from Makoto Matsumoto and Takuji Nishimura and the more recent WELL generators (Well Equidistributed

Long-period Linear) from François Panneton, Pierre L'Ecuyer and Makoto Matsumoto.

Below we show how to use the commons math API to generate different kinds of random data. The examples all use the default JDK-supplied PRNG. PRNG pluggability is covered below in a following section. The only modification required to the examples to use alternative PRNGs is to replace the argumentless constructor calls with invocations including a RandomGenerator instance as a parameter.

Random numbers

The *RandomData* interface defines methods for generating random sequences of numbers. The API contracts of these methods use the following concepts:

Random sequence of numbers from a probability distribution

There is no such thing as a single "random number." What can be generated are sequences of numbers that appear to be random. When using the built-in JDK function *Math.random()*, sequences of values generated follow the Uniform Distribution, which means that the values are evenly spread over the interval between 0 and 1, with no sub-interval having a greater probability of containing generated values than any other interval of the same length. The mathematical concept of a probability distribution basically amounts to asserting that different ranges in the set of possible values of a random variable have different probabilities of containing the value. Commons Math supports generating random sequences from each of the distributions in the distributions package. The javadoc for the *nextXxx* methods in *RandomDataImpl* describes the algorithms used to generate random deviates.

Cryptographically secure random sequences

It is possible for a sequence of numbers to appear random, but nonetheless to be predictable based on the algorithm used to generate the sequence. If in addition to randomness, strong unpredictability is required, it is best to use a secure random number generator to generate values (or strings). The *nextSecureXxx* methods in the *RandomDataImpl* implementation of the *RandomData* interface use the JDK *SecureRandom* PRNG to generate cryptographically secure sequences. The *setSecureAlgorithm* method allows you to change the underlying PRNG. These methods are much slower than the corresponding "non-secure" versions, so they should only be used when cryptographic security is required.

Seeding pseudo-random number generators

By default, the implementation provided in *RandomDataImpl* uses the JDK-provided PRNG. Like most other PRNGs, the JDK generator generates sequences of random numbers based on an initial "seed value". For the non-secure methods, starting with the same seed always produces the same sequence of values. Secure sequences started with the same seeds will diverge. When a new *RandomDataImpl* is created, the underlying random number generators are *not* initialized. The first call to a data generation method, or to a *reSeed()* method initializes the appropriate generator. If you do not explicitly seed the generator, it is by default seeded with the current time in milliseconds. Therefore, to generate sequences of random data values, you should always instantiate one *RandomDataImpl* and use it repeatedly instead of creating new instances for subsequent values in the sequence. For example, the following will generate a random sequence of 50 long integers between 1 and 1,000,000, using the current time in milliseconds as the seed for the JDK PRNG:

```
var randomData = new RandomDataImpl()
var rv = new Vec(1000)
for (i <- 0 until 1000)
  rv(i) = randomData.nextLong(1, 1000000)
plot(rv)
```

The following will not in general produce a good random sequence, since the PRG is reseeded each time through the loop with the current time in milliseconds:

```
var randomData = new RandomDataImpl()
var rv = new Vec(1000)
for (i <- 0 until 1000) {
  randomData = new RandomDataImpl()
  rv(i) = randomData.nextLong(1, 1000000)
 }
plot(rv)
```

The following will produce the same random sequence each time it is executed:

```
var randomData = new RandomDataImpl()
var rv = new Vec(1000)
randomData.reSeed(1000)
for (i <- 0 until 1000) {
  rv(i) = randomData.nextLong(1, 1000000)
 }

var rv2 = new Vec(1000)
randomData.reSeed(1000)
for (i <- 0 until 1000) {
  rv2(i) = randomData.nextLong(1, 1000000)
 }
figure(1); subplot(3,1,1); plot(rv, "rv"); subplot(3,1,2); plot(rv2, "rv2");
subplot(3,1,3); plot(rv-rv2, "rv-rv2")
```

The following will produce a different random sequence each time it is executed.

```
var randomData = new RandomDataImpl()
var rv = new Vec(1000)
randomData.reSeedSecure(1000)
for (i <- 0 until 1000) {
  rv(i) = randomData.nextLong(1, 1000000)
 }

var rv2 = new Vec(1000)
randomData.reSeedSecure(1000)
for (i <- 0 until 1000) {
  rv2(i) = randomData.nextLong(1, 1000000)
 }
figure(1); subplot(3,1,1); plot(rv, "rv"); subplot(3,1,2); plot(rv2, "rv2");
subplot(3,1,3); plot(rv-rv2, "rv-rv2")
```

Random Vectors

Some algorithms require random vectors instead of random scalars. When the components of these vectors are uncorrelated, they may be generated simply one at a time and packed together in the

vector. The *UncorrelatedRandomVectorGenerator* class simplifies this process by setting the mean and deviation of each component once and generating complete vectors. When the components are correlated however, generating them is much more difficult. The *CorrelatedRandomVectorGenerator* class provides this service. In this case, the user must set up a complete covariance matrix instead of a simple standard deviations vector. This matrix gathers both the variance and the correlation information of the probability law.

The main use for correlated random vector generation is for Monte-Carlo simulation of physical problems with several variables, for example to generate error vectors to be added to a nominal vector. A particularly common case is when the generated vector should be drawn from a *Multivariate Normal Distribution.*

The following example illustrates the generation of random vectors from a bivariate normal distribution

```
// Create and seed a RandomGenerator (could use any of the generators in the random package
here)
var rg = new JDKRandomGenerator()
rg.setSeed(17399225432l)  // Fixed seed means same results every time
// Create a GaussianRandomGenerator using rg as its source of randomness
var rawGenerator = new GaussianRandomGenerator(rg)
var mean = Array(1.0, 2.0)
var c = -2.0
var covDarr = Array( Array(9, c), Array(c, 16))
var covariance = new Array2DRowRealMatrix(covDarr)
// Create a CorrelatedRandomVectorGenerator using rawGenerator for the components
var generator = new CorrelatedRandomVectorGenerator(mean, covariance, 1.0e-12 *
covariance.getNorm(), rawGenerator)
// Use the generator to generate correlated vectors
var Nvecs = 500
var vecsAll = Array.ofDim[Double](2, Nvecs)
for (k<-0 until Nvecs) {
  var randomVector = generator.nextVector()
  vecsAll(0)(k) = randomVector(0)
  vecsAll(1)(k) = randomVector(1)
}
figure(1); subplot(3,1,1);  plot(vecsAll(0)); subplot(3,1,2); plot(vecsAll(1));
subplot(3,1,3); scatterPlotsOn; plot(vecsAll(0), vecsAll(1))
```

The *mean* argument of *CorrelatedRandomVectorGenerator()* is an Array[Double] holding the means of the random vector components. In the bivariate case, it must have length 2. The *covariance* argument is a RealMatrix, which needs to be 2 x 2. The main diagonal elements are the *variances* of the vector components and the off-diagonal elements are the covariances. For example, if the means are 1 and 2 respectively, and the desired standard deviations are 3 and 4, respectively, then we need to use

```
var mean = Array(1.0, 2.0)
var c = -2.0
var covDarr = Array( Array(9, c), Array(c, 16))
var covariance = new Array2DRowRealMatrix(covDarr)
```

where c is the desired covariance. If you are starting with a desired correlation, you need to translate this to a covariance by multiplying it by the product of the standard deviations. For example, if you

want to generate data that will give Pearson's R of 0.5, you would use c = 3 * 4 * .5 = 6.

In addition to multivariate normal distributions, correlated vectors from multivariate uniform distributions can be generated by creating a *UniformRandomGenerator* in place of the *GaussianRandomGenerator* above. More generally, any *NormalizedRandomGenerator* may be used.

Random Strings

The methods *nextHexString* and *nextSecureHexString* can be used to generate random strings of hexadecimal characters. Both of these methods produce sequences of strings with good dispersion properties. The difference between the two methods is that the second is cryptographically secure. Specifically, the implementation of *nextHexString*(n) in *RandomDataImpl* uses the following simple algorithm to generate a string of n hex digits:

1. n/2+1 binary bytes are generated using the underlying Random
2. Each binary byte is translated into 2 hex digits

The RandomDataImpl implementation of the "secure" version, *nextSecureHexString* generates hex characters in 40-byte "chunks" using a 3-step process:

1. 20 random bytes are generated using the underlying SecureRandom.
2. SHA-1 hash is applied to yield a 20-byte binary digest.
3. Each byte of the binary digest is converted to 2 hex digits.

Similarly to the secure random number generation methods, nextSecureHexString is much slower than the non-secure version. It should be used only for applications such as generating unique session or transaction ids where predictability of subsequent ids based on observation of previous values is a security concern. If all that is needed is an even distribution of hex characters in the generated strings, the non-secure method should be used.

Random permutations, combinations, sampling

To select a random sample of objects in a collection, you can use the *nextSample* method in the *RandomData* interface. Specifically, if *c* is a collection containing at least *k* objects, and randomData is a *RandomData* instance *randomData.nextSample(c, k)* will return an object[] array of length k consisting of elements randomly selected from the collection. If c contains duplicate references, there may be duplicate references in the returned array; otherwise returned elements will be unique, i.e. the *sampling is without replacement* among the object references in the collection.

If *randomData* is a *RandomData* instance, and *n* and *k* are integers with *k* <= *n*, then *randomData.nextPermutation(n, k)* returns an int[] array of length k whose entries are selected randomly, without repetition, from the integers 0 through *n-1* (inclusive), i.e., *randomData.nextPermutation(n, k)* returns a random permutation of *n* taken *k* at a time.

Generating data 'like' an input file

Using the *ValueServer* class, you can generate data based on the values in an input file in one of two ways:

Replay Mode

The following code read data from a url (a *java.net.URL* instance), cycling through the values in the file in sequence, reopening and starting at the beginning again when all values have been read.

```
var  vs = new ValueServer()
vs.setValuesFileURL(url)
vs.setMode(ValueServer.REPLAY_MODE)
vs.resetReplayFile()
var  value = vs.getNext()
// ...Generate and use more values...
vs.closeReplayFile()
```

The values in the file are not stored in memory, so it does not matter how large the file is, but you do need to explicitly close the file as above. The expected file format is \n -delimited (i.e. one per line) strings representing valid floating point numbers.

Digest Mode

When used in Digest Mode, the *ValueServer* reads the entire input file and estimates a probability density function based on data from the file. The estimation method is essentially the *Variable Kernel Method* with Gaussian smoothing. Once the density has been estimated, *getNext()* returns random values whose probability distribution matches the empirical distribution, i.e. if you generate a large number of such values, their distribution should "look like" the distribution of the values in the input file. The values are not stored in memory in this case either, so there is no limit to the size of the input file. Here is an example:

```
var  vs = new ValueServer()
vs.setValuesFileURL(url)
vs.setMode(ValueServer.DIGEST_MODE)
vs.computeDistribution(500)  //Read file and estimate distribution using 500 bins
var  value = vs.getNext()
 // ...Generate and use more values...
```

Note that *computeDistribution()* opens and closes the input file by itself.

PRNG Pluggability

To enable alternative PRNGs to be "plugged in" to the commons-math data generation utilities and to provide a generic means to replace java.util.Random in applications, a random generator adaptor framework has been added to commons-math. The *RandomGenerator* interface abstracts the public interface of java.util.Random and any implementation of this interface can be used as the source of random data for the commons-math data generation classes. An abstract base class, *AbstractRandomGenerator* is provided to make implementation easier. This class provides default implementations of "derived" data generation methods based on the primitive, *nextDouble()*. To support generic replacement of java.util.Random, the *RandomAdaptor* class is provided, which extends java.util.Random and wraps and delegates calls to a *RandomGenerator* instance.

Commons-math provides by itself several implementations of the *RandomGenerator* interface:

- *JDKRandomGenerator* that extends the JDK provided generator
- *AbstractRandomGenerator* as a helper for users generators
- *BitStreamGenerator* which is an abstract class for several generators and which in turn is extended by:
    - *Mersenne Twister*

- *Well512a*
- *Well1024a*
- *Well19937a*
- *Well19937c*
- *Well44497a*
- *Well44497b*

The JDK provided generator is a simple one that can be used only for very simple needs. The Mersenne Twister is a fast generator with very good properties well suited for Monte-Carlo simulation. It is equidistributed for generating vectors up to dimension 623 and has a huge period: 2^{19937} - 1 (which is a Mersenne prime). This generator is described in a paper by Makoto Matsumoto and Takuji Nishimura in 1998: *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, January 1998, pp 3--30. The WELL generators are a family of generators with period ranging from 2^{512} - 1 to 2^{44497} - 1 (this last one is also a Mersenne prime) with even better properties than Mersenne Twister. These generators are described in a paper by François Panneton, Pierre L'Ecuyer and Makoto Matsumoto *Improved Long-Period Generators Based on Linear Recurrences Modulo 2 ACM Transactions on Mathematical Software*, 32, 1 (2006).

For simple sampling, any of these generators is sufficient. For Monte-Carlo simulations the JDK generator does not have any of the good mathematical properties of the other generators, so it should be avoided. The Mersenne twister and WELL generators have equidistribution properties proven according to their bits pool size which is directly linked to their period (all of them have maximal period, i.e. a generator with size *n* pool has a period 2^n-1). They also have equidistribution properties for 32 bits blocks up to s/32 dimension where s is their pool size. So WELL19937c for example is equidistributed up to dimension 623 (19937/32). This means a Monte-Carlo simulation generating a vector of *n* variables at each iteration has some guarantees on the properties of the vector as long as its dimension does not exceed the limit. However, since we use bits from two successive 32 bits generated integers to create one double, this limit is smaller when the variables are of type double, so for Monte-Carlo simulation where less the 16 doubles are generated at each round, WELL1024 may be sufficient. If a larger number of doubles are needed a generator with a larger pool would be useful.

The WELL generators are more modern than MersenneTwister (the paper describing them has been published in 2006 instead of 1998) and fix some of its (few) drawbacks. If initialization array contains many zero bits, MersenneTwister may take a very long time (several hundreds of thousands of iterations to reach a steady state with a balanced number of zero and one in its bits pool). So the WELL generators are better to *escape zeroland* as explained by the WELL generators creators. The Well19937a and Well44497a generator are not maximally equidistributed (i.e. there are some dimensions or bits blocks size for which they are not equidistributed). The Well512a, Well1024a, Well19937c and Well44497b are maximally equidistributed for blocks size up to 32 bits (they should behave correctly also for double based on more than 32 bits blocks, but equidistribution is not proven at these blocks sizes).

The MersenneTwister generator uses a 624 elements integer array, so it consumes less than 2.5 kilobytes. The WELL generators use 6 integer arrays with a size equal to the pool size, so for example the WELL44497b generator uses about 33 kilobytes. This may be important if a very large number of generator instances were used at the same time.

All generators are quite fast. As an example, here are some comparisons, obtained on a 64 bits JVM on a Linux computer with a 2008 processor (AMD phenom Quad 9550 at 2.2 GHz). The generation rate for MersenneTwister was between 25 and 27 millions doubles per second (remember we generate two 32 bits integers for each double). Generation rates for other PRNG, relative to

MersenneTwister:

Example of performances	
Name	**generation rate (relative to MersenneTwister)**
MersenneTwister	1
JDKRandomGenerator	between 0.96 and 1.16
Well512a	between 0.85 and 0.88
Well1024a	between 0.63 and 0.73
Well19937a	between 0.70 and 0.71
Well19937c	between 0.57 and 0.71
Well44497a	between 0.69 and 0.71
Well44497b	between 0.65 and 0.71

So for most simulation problems, the better generators like Well19937c and Well44497b are probably very good choices.

Note that *none* of these generators are suitable for cryptography. They are devoted to simulation, and to generate very long series with strong properties on the series as a whole (equidistribution, no correlation, etc). They do not attempt to create small series but with very strong properties of unpredictability as needed in cryptography.

Examples:

Create a *RandomGenerator* based on RngPack's Mersenne Twister
To create a RandomGenerator using the RngPack Mersenne Twister PRNG as the source of randomness, extend *AbstractRandomGenerator* overriding the derived methods that the RngPack implementation provides:

```
import edu.cornell.lassp.houle.RngPack.RanMT
// AbstractRandomGenerator based on RngPack RanMT generator.
class RngPackGenerator extends AbstractRandomGenerator {
  var  random = new RanMT()
  override def  setSeed(seed:Long): Unit =  {
    random = new RanMT(seed)
  }
  override def  nextDouble() = {
    random.raw()
    }
  override  def  nextGaussian() = {
    random.gaussian()
    }
  override def  nextInt(n: Int) = {
    random.choose(n)
    }
  override def nextBoolean() = {
```

```
    random.coin()
    }
}
```

Use the Mersenne Twister RandomGenerator in place of java.util.Random in RandomData:

```
var randomData = new RandomDataImpl(new RngPackGenerator())
```

Create an adaptor instance based on the Mersenne Twister generator that can be used in place of a Random:

```
var generator = new RngPackGenerator()
var random = RandomAdaptor.createAdaptor(generator)
// random can now be used in place of a Random instance, data generation calls
// will be delegated to the wrapped Mersenne Twister
```

Linear Algebra

Overview

Linear algebra support in commons-math provides operations on real matrices (both dense and sparse matrices are supported) and vectors. It features basic operations (addition, subtraction, etc) and decomposition algorithms that can be used to solve linear systems either in exact sense and in least squares sense.

Real matrices

The *RealMatrix* interface represents a matrix with real numbers as entries. The following basic matrix operations are supported:

- Matrix addition, subtraction, multiplication
- Scalar addition and multiplication
- transpose
- Norm and Trace
- Operation on a vector

Example:

```
// Create a real matrix with two rows and three columns
var matrixData = Array(Array(1.0, 2.0, 3.0), Array(2.0, 5.0, 3.0))
var m = new Array2DRowRealMatrix(matrixData)
// One more with three rows, two columns
var matrixData2 = Array( Array(1.0, 2.0), Array(2.0, 5.0), Array(1.0, 7.0))
var n = new Array2DRowRealMatrix(matrixData2)
// Note: The constructor copies the input double[][] array.
// Now multiply m by n
var p = m.multiply(n)
println(p.getRowDimension())  // 2
println(p.getColumnDimension()) // 2
// Invert p, using LU decomposition
var pInverse = new LUDecompositionImpl(p).getSolver().getInverse()
```

The three main implementations of the interface are *Array2DRowRealMatrix* and *BlockRealMatrix*

for dense matrices (the second one being more suited to dimensions above 50 or 100) and *SparseRealMatrix* for sparse matrices.

Real Vectors

The *RealVector* interface represents a vector with real numbers as entries. The following basic matrix operations are supported:

- Vector addition, subtraction
- Element by element multiplication, division
- Scalar addition, subtraction, multiplication, division and power
- Mapping of mathematical functions (cos, sin, etc.)
- Dot product, outer product
- Distance and norm according to norms L1, L2 and Linf

The *RealVectorFormat* class handles input/output of vectors in a customizable textual format.

Solving linear systems

The *solve()* methods of the *DecompositionSolver* interface support solving linear systems of equations of the form AX=B, either in the linear sense or in the least square sense. A *RealMatrix* instance is used to represent the coefficient matrix of the system. Solving the system is a two phases process: first the coefficient matrix is decomposed in some way and then a solver built from the decomposition solves the system. This allows to compute the decomposition and build the solver only once if several systems have to be solved with the same coefficient matrix. For example, to solve the linear system

2x + 3y - 2z = 1

-x + 7y + 6x = -2

4x - 3y - 5z = 1

Start by decomposing the coefficient matrix A (in this case using LU decomposition) and build a solver:

var coefficients = new Array2DRowRealMatrix(Array(Array(2.0, 3.0, -2), Array(-1.0, 7.0, 6), Array(4, -3.0, -5)))

var solver = new LUDecompositionImpl(coefficients).getSolver()

Next create a *RealVector* array to represent the constant vector B and use *solve(RealVector)* to solve the system

var constants = new ArrayRealVector(Array(1.0, -2.0, 1), false)
var solution = solver.solve(constants)

The *solution* vector will contain values for *x (solution.getEntry(0))*, *y(solution.getEntry(1))*, and *z (solution.getEntry(2))* that solve the system.

Each type of decomposition has its specific semantics and constraints on the coefficient matrix as shown in the following table. For algorithms that solve AX=B in least squares sense the value returned for X is such that the residual AX-B has minimal norm. If an exact solution exist (i.e. if for some X the residual AX-B is exactly 0), then this exact solution is

also the solution in least square sense. This implies that algorithms suited for least squares problems can also be used to solve exact problems, but the reverse is not true.

Decomposition algorithm	Coefficients Matrix	problem type
LU	square	exact solution only
Cholesky	symmetric positive definite	exact solution only
QR	any	least squares solution
eigen decomposition	square	exact solution only
SVD	any	least squares

It is possible to use a simple array of double instead of a *RealVector.* In this case, the solution will be provided also as an array of double.

It is possible to solve multiple systems with the same coefficient matrix in one method call. To do this, create a matrix whose column vectors correspond to the constant vectors to be solved and use *solve(RealMatrix)* , which returns a matrix with column vectors representing the solutions.

Eigenvalues/eigenvectors and singular values/singular vectors

Decomposition algorithms may be used for themselves and not only for linear system solving. This is of prime interest with eigen decomposition and singular value decomposition.

The *getEigenvalue(), getEigenvalues(), getEigenVector(), getV(), getD()* and *getVT()* methods of the *EigenDecomposition* interface support solving eigenproblems of the form *AX = lambda X* where *lambda* is a real scalar. The *getSingularValues(), getU(), getS()* and *getV()* methods of the *SingularValueDecomposition* interface allow to solve singular values problems.

Ordinary Differential Equations

Overview

The ode package of Apache Commons provides classes to solve Ordinary Differential Equations problems. ScalaLab provides routines to solve ODEs with the NUMAL library [11], the Numerical Recipes implementations [12], and thus the Apache Commons is an alternative option. We should note that Apache Commons supports a wide range of ODE problems.

This package solves Initial Value Problems of the form *y'=f(t,y)* with *t0* and *y(t0)=y0* known. The provided integrators compute an estimate of *y(t)* from *t=t0* to *t=t1*.

Also integrators provide *dense output*. This means that besides computing the state vector at discrete times, they also provide a cheap mean to get both the state and its derivative between the time steps. They do so through classes extending the *StepInterpolator* abstract class, which are made available to the user at the end of each step.

All integrators handle multiple discrete events detection based on switching functions. This means that the integrator can be driven by user specified discrete events (occurring when the sign of a user-supplied *switching function* changes). The steps are shortened as needed to ensure the events occur at step boundaries (even if the integrator is a fixed-step integrator). When the events are triggered, integration can be stopped (this is called a *G-stop facility*), the state vector can be changed, or integration can simply go on. The latter case is useful to handle discontinuities in the differential equations gracefully and get accurate dense output even close to the discontinuity.

All integrators support setting a maximal number of evaluations of the differential equations function. If this number is exceeded, an exception will be thrown during integration. This can be used to prevent infinite loops if for example error control or discrete events create a really large number of extremely small steps. By default, the maximal number of evaluation is set to Integer.MAX_VALUE (i.e. 2^31-1 or 2147483647). It is recommended to set this maximal number to a value suited to the ODE problem, integration range, and step size or error control settings.

The user should describe the problem in classes which should implement the *FirstOrderDifferentialEquations* interface. Then the description should be passed to the preferred integrator among all the classes that implement the *FirstOrderIntegrator* interface. The following example shows how to implement the simple two-dimensional problem:

$$\dot{y}_0(t) = \omega(c_1 - y_1(t))$$
$$\dot{y}_1(t) = \omega(y_0(t) - c_0)$$

with some initial state $y(t_0) = (y0(t_0), y1(t_0))$. In fact, the exact solution of this problem is that y(t) moves along a circle centered at c = (c0, c1) with constant angular rate ω.

We present the class *CircleODE* that implements our differential equation below.

```
class CircleODE( c: Array[Double], omega: Double) extends AnyRef with
FirstOrderDifferentialEquations {
  def getDimension = 2
  def computeDerivatives(t: Double, y: Array[Double], yDot: Array[Double]) = {
        yDot(0) = omega*(c(1) - y(1))
        yDot(1) = omega*(y(0)-c(0))
        }
 }
```

Computing the state y(16.0) starting from y(0.0) = (0.0, 1.0) and integrating the ODE is done as follows (using the Dormand-Prince 8(5,3) integrator as an example):

```
var  dp853 = new DormandPrince853Integrator(1.0e-8, 100.0, 1.0e-10, 1.0e-10)
```

```
var ode = new CircleODE( Array(1.0, 1.0), 0.1)
var y = Array(0.0, 1.0) // initial state
dp853.integrate(ode, 0.0, y, 16.0, y)

y // now y contains final state at time t = 16.0
```

The Dormand-Prince integrator has the following constructor interface:

```
public DormandPrince853Integrator(final double minStep, final double maxStep,

final double scalAbsoluteTolerance,  final double scalRelativeTolerance)
```

therefore the parameters above are instantiated as: minStep=1.0e-8, maxStep=100.0, scalAbsoluteTolerance=1.0e-10, scalRelativeTolerance=1.0e-10

Continuous Output

The solution of the integration problem is provided by two means. The first one is aimed towards simple use: the state vector at the end of the integration process is copied in the y array of the *FirstOrderIntegrator.integrate* method, as shown by the previous example. The second one should be used when more in-depth information is needed throughout the integration process. The user can register an object implementing the *StepHandler* interface or a *StepNormalizer* object wrapping a user-specified object implementing the *FixedStepHandler* interface into the integrator before calling the *FirstOrderIntegrator.integrate* method. The user object will be called appropriately during the integration process, allowing the user to process intermediate results. The default step handler does nothing. Considering again the previous example, we want to print the trajectory of the point to check that it really is a circle arc. We simply add the following before the call to *integrator.integrate*:

```
def stepHandler = new StepHandler() {
        def reset= {}
        def requiresDenseOutput = false
        def handleStep(interpolator: StepInterpolator, isLast: Boolean) =
         {
         var t = interpolator.getCurrentTime
        var y = interpolator.getInterpolatedState
        println(t+" "+y(0)+" "+y(1))
         }
        }
integrator.addStepHandler(stepHandler)
```

ContinuousOutputModel is a special-purpose step handler that is able to store all steps and to provide transparent access to any intermediate result once the integration is over. An important feature of this class is that it implements the *Serializable* interface. This means

that a complete continuous model of the integrated function throughout the integration range can be serialized and reused later (if stored into a persistent medium like a file system or a database) or elsewhere (if sent to another application). Only the result of the integration is stored, there is no reference to the integrated problem by itself.

Other default implementations of the *StepHandler* interface are available for general needs (*DummyStepHandler, StepNormalizer*) and custom implementations can be developed for specific needs. As an example, if an application is to be completely driven by the integration process, then most of the application code will be run inside a step handler specific to this application.

Some integrators (the simple ones) use fixed steps that are set at creation time. The more efficient integrators use variable steps that are handled internally in order to control the integration error with respect to a specified accuracy (these integrators extend the *AdaptiveStepsizeIntegrator* abstract class). In this case, the step handler which is called after each successful step shows up the variable stepsize. The *StepNormalizer* class can be used to convert the variable stepsize into a fixed stepsize that can be handled by classes implementing the *FixedStepHandler* interface. Adaptive stepsize integrators can automatically compute the initial stepsize by themselves, however the user can specify it if prefers to retain full control over the integration or if the automatic guess is wrong.

Discrete Events Handling

ODE problems are continuous ones. However, sometimes discrete events must be taken into account. The most frequent case is when the stop condition of the integrator is not defined by the time *t* but by a target condition on state *y* (say y(0) = 1.0 for example).

Discrete events detection is based on `switching functions`. The user provides a simple *g(t, y)* function depending on the *current time and state*. The integrator will monitor the value of the function throughout integration range and will trigger the event when its sign changes. The magnitude of the value is almost irrelevant, it should however be continuous (but not necessarily smooth) for the sake of root finding. The steps are shortened as needed to ensure the events occur at step boundaries (even if the integrator is a fixed-step integrator). Note that g function signs changes at the very beginning of the integration (from t0 to t0 + ε where ε is the events detection convergence threshold) are explicitly ignored. This prevents having the integration stuck at its initial point when a new integration is restarted just at the same point a previous one had been stopped by an event.

When an event is triggered, the event time, current state and an indicator whether the switching function was increasing or decreasing at event time are provided to the user. Several different options are available:

1. integration can be stopped (this is called a G-stop facility),
2. the state vector or the derivatives can be changed,
3. or integration can simply go on.

The first case, G-stop, is the most common one. A typical use case is when an ODE must be solved up to some reached target state, with a known value of the state but an unknown occurrence time. As an example, if we want to monitor a chemical reaction up to some predefined concentration for the first substance, we can use the following switching function setting:

```
def g(t: Double, y: Array[Double] ) =  {
    y(0) – targetConcentration
}
def eventOccurred(t: Double, y: Array[Double], increasing: Boolean) = STOP
```

The second case, changing state vector or derivatives is encountered when dealing with discontinuous dynamical models. A typical case would be the motion of a spacecraft when thrusters are fired for orbital maneuvers. The acceleration is smooth as long as no maneuvers are performed, depending only on gravity, drag, third body attraction, radiation pressure. Firing a thruster introduces a discontinuity that must be handled appropriately by the integrator. In such a case, we would use a switching function setting similar to this:

```
def g(t: Double, y: Array[Double]) =  {
   (t  -  tManeuverStart) * (t – tManeuverStop)
}

def eventOccurred(t: Double, y: Array[Double], increasing: Boolean) = RESET_DERIVATIVES
```

The third case is useful mainly for monitoring purposes, a simple example is:

```
def g(t: Double, y: Array[Double]) =  y(0) – y(1)

def  eventOccurred(t: Double, y: Array[Double], increasing: Boolean) = {
   logger.log("y0(t) and y1(t) curves cross at t = "+t)

}
```

Available Integrators

The tables below show the various integrators available for non-stiff problems. Note that the implementations of Adams-Bashforth and Adams-Moulton are adaptive stepsize, not fixed stepsize as is usual for these multi-step integrators. This is due to the fact that the implementation relies on the Nordsieck vector representation of the state.

Fixed Step Integrators	
Name	Order
Euler	1
Midpoint	2
Classical Runge-Kutta	4
Gill	4
3 / 8	4
Luther	6

Adaptive Stepsize Integrators		
Name	Integration Order	Error Estimation Order
Higham and Hall	5	4
Dormand-Prince 5(4)	5	4
Dormand-Prince 8(5,3)	8	5 and 3
Gragg-Bulirsch-Stoer	variable (up to 18 by default)	variable
Adams-Bashforth	variable	variable
Adams-Moulton	variable	variable

Derivatives

If in addition to state *y(t)* the user needs to compute the sensitivity of the state to the initial state or some parameter of the ODE, can use the classes and interfaces from the *org.apache.commons.ode.jacobians* package instead of the top level ode package. These classes compute the jacobians $dy(t)/dy_0$ and $dy(t)/dp$ where y_0 is the initial state and *p* is some ODE parameter.

The classes and interfaces in this package mimic the behavior of the classes and interfaces of the top level ode package, only adding parameters arrays for the jacobians. The behavior of these classes is to create a compound state vector z containing both the state y(t) and its derivatives $dy(t)/dy_0$ and $dy(t)/dp$ and to set up an extended problem by adding the equations for the jacobians automatically. These extended state and problems are then provided to a classical underlying integrator chosen by the user.

This behavior imply that there will be a top level integrator knowing about state and jacobians and a low level integrator knowing only about compound state (which may be big). If the user wants to deal with the top level only, should use the specialized step handler and event handler classes registered at the top level. The user can also register classical step handlers and event handlers, but in this case will see the big compound state. This state is guaranteed to contain the original state in the first elements, followed by the jacobian with respect to initial state (in row order), followed by the jacobian with respect to parameters (in row order). If for example the original state dimension is 6 and there are 3 parameters, the compound state will be an array with 60 elements. The first 6 elements will be the original state, the next 36 elements will be the jacobian with respect to initial state, and the remaining 18 will be the jacobian with respect to parameters. Dealing with low level step handlers and event handlers is cumbersome if one really needs the jacobians in these methods, but it also prevents many data being copied back and forth between state and jacobians on one side and compound state on the other side.

In order to compute $dy(t)/dy_0$ and $dy(t)/dp$ for any t, the algorithm needs not only the ODE function f such that *y'=f(t,y)* but also its local jacobians *df(t, y, p)/dy* and *df(t, y, p)/dp*.

If the function f is too complex, the user can simply rely on internal differentiation using finite differences to compute these local jacobians. So rather than the *FirstOrderDifferentialEquations* interface, the *ParameterizedODE* interface should be implemented. Considering again our example where only ω is considered a parameter, we get:

```
class  BasicCircleODE(val c: Array[Double], var omega: Double) extends AnyRef with
ParameterizedODE  {
        def  getDimension = 2
        def computeDerivatives(t: Double, y: Array[Double], yDot: Array[Double]) = {
                yDot(0) = omega*(c(1) - y(1))
                yDot(1) = omega*(y(0)-c(0))
        }
        def getParametersDimension = {
  // we are only interested in the omega parameters
        1
        }
        def setParameter(i: Int, value: Double) = { omega = value }
}
```

This ODE is provided to the specialized integrator with two arrays specifying the step sizes to use for finite differences (one array for derivation with respect to state y, one array for derivation with respect to parameters p):

```
var  hY = Array(0.01, 0.01)
var hP = Array(1.0e-6)
var intergrator = new FirstOrderIntegratorWithJacobians(dp853, ode, hY, hP)
integrator.integrate(t0, y0, dy0dp, t, y, dydy0, dydp)
```

If the function f is simple, the user can simply provide the local jacobians. So rather than the *FirstOrderDifferentialEquations* interface the implementation of the *ODEWithJacobians* interface is required. Considering again our example where only ω is considered a parameter, we get:

```
class EnchancedCircleODE  (val c: Array[Double], var omega: Double) extends AnyRef with
ODEwithJacobians  {

        def  getDimension = 2
        def computeDerivatives(t: Double, y: Array[Double], yDot: Array[Double]) = {
                yDot(0) = omega*(c(1) - y(1))
                yDot(1) = omega*(y(0)-c(0))
        }
        def getParametersDimension = {
  // we are only interested in the omega parameters
        1
        }
        def setParameter(i: Int, value: Double) = { omega = value }

        def  computeJacobians(t:  Double,  yDot:  Array[Double],  dFdY:
Array[Array[Double]],
```

```
                dFdP: Array[Double]) =  {
            dFdY(0)(0) = 0
            dFdY(0)(1) = -omega
            dFdY(1)(0) = omega
            dFdY(1)(1) = 0


            dFdP(0)(0) = 0
            dFdP(0)(1) = omega
            dFdY(1)(2) = c(1)-y(1)
            dFdP(1)(0) = -omega
            dFdP(1)(1) = 0
            dFdP(1)(2) = y(0)-c(0)
    }
}
```

This ODE is provided to the specialized integrator as is:

```
var integrator = new FirstOrderIntegratorWithJacobians(dp853, ode)
integrator.integrate(t0, y0, dy0dp, t, y, dydy0, dydp)
```

Polynomial Fitting Example

We present an example of using the Apache Commons Math library to fit data with a polynomial model. The data are generated synthetically from a known function (that is a 2nd degree polynomial) and noisy is injected.

Polynomial fitting code

The function *avalue* is the function with which the data are generated. We inject some amount of uniformly distributed noise to that data.

Next, we proceed to recover the generating function, knowing however its model, i.e. that is a 2nd degree polynomial. In order to perform the recovery we use the Levenberg Marquardt Optimizer from the Apache Commons Maths library. The plot demonstrates that the algorithm succeeds in accurately recovering the parameters of the polynomial from the noise measurements.

Note that in order to run the following code we should import the Apache Common Maths, which can be done easily with the *importApacheCommons* helper method. The top level menu *Imports* of the ScalaLab GUI facilitates importing from libraries.

```
importApacheCommons   // import some significant Apache Common Math classes
def avalue(x: Double) = 4.5*x*x - 3.7*x+5.6
```

```
var nP = 20
var x = linspace(0, 2, nP)
var ay = new Array[Double](nP)
var y = new Array[Double](nP)
  // generate synthetic data points
for (k<-0 until nP) {
 ay(k) = avalue(x(k))   // function value
 y(k) = ay(k)+(java.lang.Math.random()-0.5)   // induce some noise
}
var optimizer = new LevenbergMarquardtOptimizer()
var weights = new Array[Double](nP)
for (k<-0 until nP) weights(k)=1.0
var initialSolution = Array(1.0, 1.0, 1.0)
var optProblem = new PolynomialProblem(x, ay) // PolynomialProblem is our class, defined below
var optimum = optimizer.optimize(100, optProblem, y, weights, initialSolution)
var oV = optimum.getPoint
var yrecover = new Array[Double](nP)
for (k<-0 until nP)
 yrecover(k) = oV(0)*x(k)*x(k)+oV(1)*x(k)+oV(2)
hold("on")
linePlotsOn
plot(x, yrecover, Color.BLUE)
plot(x,ay)
scatterPlotsOn
plot(x, y, Color.GREEN)


// constructs a 2nd degree polynomial that fits the x and y values
class PolynomialProblem( x: Array[Double], y: Array[Double])
 extends AnyRef with DifferentiableMultivariateVectorialFunction {

  def jacobian( variables: Array[Double]) = {
    var jac =  Array.ofDim[Double](x.length, 3)
    for (i <- 0 until x.length) {
      jac(i)(0) = x(i)*x(i)
      jac(i)(1) = x(i)
      jac(i)(2) = 1.0
    }
    jac
    }

  def value( variables: Array[Double]) = {
    var values = new Array[Double](x.length)
    for (i <- 0 until values.length)
     values(i) = (variables(0)*x(i)+variables(1))*x(i) + variables(2)
   values
   }

  def jacobian() = {
    new MultivariateMatrixFunction() {
      def value( point: Array[Double] ) =
       jacobian( point )
```

```
    }
   }
 }
```

Testing a fit both with a correct degree polynomial and a higher-order polynomial than the actual (i.e. illustrating overfitting)

The example below demonstrates that the fitting of a higher order polynomial fails on testing data.

```
importApacheCommons   // import some significant Apache Common Math classes

// the model function of the data. It is used to generate noise samples.
def  avalue(x: Double) =  4.5*x*x -  3.7*x+5.6

var nPtrain  = getInt("Number of points to fit the polynomial", 4)  // number of training set samples
var nPtest  = getInt("Number of points to test the fitted polynomial", 20)  // number of testing set samples

var x = linspace(0, 2, nPtrain)
var ay = new Array[Double](nPtrain)    // the actual values
var dy = new Array[Double](nPtrain)    // the measured data values, i.e. the actual values
                                       // with the injected noise

for (k<-0 until nPtrain) {
  ay(k) = avalue(x(k))  // actual value
  dy(k) = ay(k)+4*(Math.random-0.5)   // add some noise for the measured data value
}

// construct  test points
var tx = linspace(2, 4, nPtest)
var aty = new Array[Double](nPtest)   // the actual values for test points

for (k<-0 until nPtest)
```

```
  aty(k) = avalue(tx(k))  // actual value for test points


var cmy = new Array[Double](nPtest)   // the correct model predicted values


//var useLU = true; var optimizer = new GaussNewtonOptimizer(useLU); var figTitle = "Gauss-Newton Optimizer"


var optimizer = new LevenbergMarquardtOptimizer(); var figTitle = "Levenberg-Marquardt Optimizer"


// weight vector accounting significant of points

var weights = new Array[Double](nPtrain)

for (k<-0 until nPtrain) weights(k)=1.0

var initialSolutionCorrectModel = Array(1.0, 1.0, 1.0)   // initial solution for the uknown parameters for the correct model

var initialSolutionComplexModel = Array(1.0, 1.0, 1.0, 1.0)   // initial solution for the uknown parameters for the complex model



//  construct an optimization problem using the correct model of the data

var  optCorrectModelProblem = new PolynomialProblem(x, dy)



//  construct an optimization problem using a more complex model of the data

var  optComplexModelProblem = new HOPolynomialProblem(x, dy)


var maxEval = 100

var optimumCorrectModel = optimizer.optimize(maxEval, optCorrectModelProblem, dy, weights, initialSolutionCorrectModel)

var optimumComplexModel = optimizer.optimize(maxEval, optComplexModelProblem, dy, weights, initialSolutionComplexModel)


var oV = optimumCorrectModel.getPoint  // get correct model parameters

var oVCM = optimumComplexModel.getPoint    // get complex model parameters



// evaluate models on train data
```

```
var yrecoverTrainCorrectModel = new Array[Double](nPtrain)
for (k<-0 until nPtrain)
  yrecoverTrainCorrectModel(k) = oV(0)*x(k)*x(k)+oV(1)*x(k)+oV(2)

var yrecoverTrainComplexModel = new Array[Double](nPtrain)
for (k<-0 until nPtrain)
  yrecoverTrainComplexModel(k) = oVCM(0)*x(k)*x(k)*x(k)+oVCM(1)*x(k)*x(k)+oVCM(2)*x(k)
+oVCM(3)



figure(1)
hold("on")
linePlotsOn
plot(x, yrecoverTrainCorrectModel, "Correct Model", Color.BLUE )
plot(x, yrecoverTrainComplexModel, "Complex Model", Color.GRAY)
plot(x,ay,  "Actual", Color.RED)

scatterPlotsOn
plot(x, dy, "Measured points", Color.GREEN)

title("Fig 1 : Fitting on Training Data :  "+figTitle)

//  evaluate models on test data
var yactualData = new Array[Double](nPtest)
for (k<-0 until nPtest)
  yactualData(k) = avalue(tx(k))

var yrecoverTestCorrectModel = new Array[Double](nPtest)
for (k<-0 until nPtest)
  yrecoverTestCorrectModel(k) = oV(0)*tx(k)*tx(k)+oV(1)*tx(k)+oV(2)

var yrecoverTestComplexModel = new Array[Double](nPtest)
for (k<-0 until nPtest)
  yrecoverTestComplexModel(k) = oVCM(0)*tx(k)*tx(k)*tx(k)+oVCM(1)*tx(k)*tx(k)
+oVCM(2)*tx(k)+oVCM(3)
```

```
figure(2)
hold("on")
plot(tx, yactualData, "Actual Data", Color.RED)
linePlotsOn
plot(tx, yrecoverTestCorrectModel, "Recovered with Correct Model", Color.BLUE)
plot(tx, yrecoverTestComplexModel, "Recovered with Complex Model", Color.GRAY)

title("Fig 2 : Generalization Performance : "+figTitle)

//  construct a Polynomial model with the correct order, i.e. 3nd degree
class PolynomialProblem( x: Array[Double], y: Array[Double])
  extends AnyRef with DifferentiableMultivariateVectorialFunction {

   def jacobian( variables: Array[Double]) = {
      var jac = Array.ofDim[Double](x.length, 3)
      for (i <- 0  until x.length) {
         jac(i)(0) = x(i)*x(i)
         jac(i)(1) = x(i)
         jac(i)(2) = 1.0
       }
       jac
     }

    def  value( variables: Array[Double]) = {
       var values = new Array[Double](x.length)
       for (i <- 0 until values.length)
         values(i) = (variables(0)*x(i)+variables(1))*x(i) + variables(2)
      values
      }

    def jacobian() = {
       new MultivariateMatrixFunction() {
          def value( point: Array[Double] ) =
            jacobian( point )
           }
```

```
        }
    }



// construct a Higher-Order Polynomial model than the correct order, i.e. a fourth degree instead of three
class HOPolynomialProblem( x: Array[Double], y: Array[Double])
  extends AnyRef with DifferentiableMultivariateVectorialFunction {


// the jacobian of the 4th-degree polynomial
   def jacobian( variables: Array[Double]) = {
      var jac = Array.ofDim[Double](x.length, 4)
      for (i <- 0  until x.length) {
         jac(i)(0) = x(i)*x(i)*x(i)
         jac(i)(1) = x(i)*x(i)
         jac(i)(2) = x(i)
         jac(i)(3) = 1.0
       }
      jac
    }


// the evaluated value of the 4th-degree polynomial
    def  value( variables: Array[Double]) = {
       var values = new Array[Double](x.length)
       for (i <- 0 until values.length)
         values(i) = variables(0)*x(i)*x(i)*x(i)+variables(1)*x(i)*x(i) + variables(2)*x(i) + variables(3)
      values
      }


    def jacobian() = {
       new MultivariateMatrixFunction() {
          def value( point: Array[Double] ) =
            jacobian( point )
           }
```

```
    }
}
```

Wavelets in ScalaLab

Wavelet Theory has been emerged in recent years and has found a lot of applications. It still remains a hot research topic.

In ScalaLab we can experiment easily with Wavelets by using either the wavelet package of the jSci library or the jWave toolbox which is available for installation. Here we present some examples of using these libraries.

Example 1 - Continuous Wavelet Transform Example

The Continuous Wavelet Transform example illustrates that ScalaLab interfaces easily with Java libraries of scientific code.
In particular we interface with an implementation of continuous Morlet Wavelet Transform provided from Richard Buessow (buessow@tu.berlin.de). The CWT method is:
CWT(double []y, int f, double fmax, int maxNf, String linlog, int stepfac, int df0)
where, *fs* the *sampling frequency*, *fmax* the maximum frequency of the analysis, *maxNf* the maximum number of frequencies for which analysis is performed, *stepfac:* controls the change of the step size, i.e. “frequency adaptive” change at step, and *df0* the starting frequency *f0.*
The interested reader can study this implementation by downloading ScalaLab's source code. The particular Java code that implements the Continuous Wavelet Transform is within the package wavelets, in the source file CWT.java.

The following example demonstrates performing a Continuous Wavelet Transform (CWT). The CWT class is included with the standard ScalaLab libraries. The toolbox **jWave** should not be

installed in order to run the following code, since there are name collisions.

```
var fs = 600; // 2050;

var dt = 1.0/fs
var t = inc(1, dt, 2)
var PI = 3.1415926
var PI2 = 2*PI
var y = sin(PI2*10*t)+4*cos(PI2*4*t)
var rv = rand(y.length) // a random vector of the same size as y
var summedVec = 4*y+rv
figure(1); plot(summedVec); title("Signal and Noise")
var fstart = 1 // frequency to start
var fmax = fs/2.0
var maxNf = 20
var linlog = "log"
var stepfac=16
var df0=3
var ycwt = new wavelets.CWT(y.getv, fs, fmax, maxNf, linlog, stepfac, df0)
var ed = ycwt.ed() // energy density coefficients as a double[][] vector
figure(2); plot(ed); title("Continuous Wavelet Transform")

var edm = new Matrix(ed);
//subsampledEdm = edm.resample(5, 1); // subsample matrix before displaying it in
contour plot
figure(1);
subplot(2,1,1);
plot(y); title("signal");
subplot(2,1,2);
plot2d_scalogram(ed, "scalogram");
```

Example 1 - Daubechies wavelet with the jWave toolbox

After downloading and installing the Wavelet toolbox, **jWave.jar**, you can execute the following example code:

```
// import the relevant material from the jWave toolbox
import _root_.math.transform.jwave.handlers.wavelets._
import _root_.math.transform.jwave.handlers._
// create a Daubechies Wavelet
var daubWav = new Daub03
// create a synthetic signal and inject to it some noise
var N=pow(2, 14.0).toInt
var F1 = 23.4; var F2 = 0.45; var F3= 9.8;
var taxis = linspace(0, 5, N)
var sig = 2.5*sin(F1*taxis)+8.9*cos(F2*taxis)-9.8*sin(F3*taxis)
var rndSig = vrand(N)
```

```
var sigAll = (sig+rndSig).getv
// create a FWT object
var fwtObj = new FastWaveletTransform(daubWav)
// perform a Fast Wavelet Transform with the Daubechies Wavelet
var transfSig = fwtObj.forwardWavelet(sigAll)
// plot the results
figure(1); subplot(2,1,1); plot(sigAll); title("A Signal with Noise");
subplot(2,1,2); plot(transfSig); title("Wavelet Transformed")
```

Example for 1-D DFT

```
import _root_.math.transform.jwave.Transform

import _root_.math.transform.jwave.handlers.DiscreteFourierTransform
var t = new Transform( new DiscreteFourierTransform())
var arrTime = Array(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
var arrFreq = t.forward(arrTime)  // 1-D DFT forward
var arrReco = t.reverse(arrFreq)
```

Example for 1-D FWT, 2-D FWT

```
import _root_.math.transform.jwave.Transform
import _root_.math.transform.jwave.handlers.wavelets._
import _root_.math.transform.jwave.handlers.FastWaveletTransform
var t = new Transform( new FastWaveletTransform(new Haar02()))
var arrTime = Array(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
var arrHilb = t.forward(arrTime)  // 1-D FWT Haar forward
var arrReco = t.reverse(arrHilb) // 1-D FWT Haar reverse
var matTime = Array(Array( 1., 1., 1., 1.), Array( 1., 1., 1., 1. ), Array(1., 1., 1., 1. ), Array( 1., 1.,
1., 1. ))
var matHilb = t.forward( matTime ); // 2-D FWT Haar forward
var matReco = t.reverse( matHilb ); // 2-D FWT Haar reverse
```

Example for Wavelet Packet Transform

```
import _root_.math.transform.jwave.Transform
import _root_.math.transform.jwave.handlers.wavelets._
import _root_.math.transform.jwave.handlers.WaveletPacketTransform
var t = new Transform( new WaveletPacketTransform(new Haar02()))
var arrTime = Array(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
var arrHilb = t.forward(arrTime)  // 1-D FWT Haar forward
var arrReco = t.reverse(arrHilb) // 1-D FWT Haar reverse
var matTime = Array(Array( 1., 1., 1., 1.), Array( 1., 1., 1., 1. ), Array(1., 1., 1., 1. ),
Array( 1., 1., 1., 1. ))
var matHilb = t.forward( matTime ); // 2-D WPT Haar forward
var matReco = t.reverse( matHilb ); // 2-D WPT Haar reverse
```

BioScala in ScalaLab

Scala is a great language to perform Computational Biology and Bioinformatics tasks. The BioScala toolbox of ScalaLab is one way of starting exploring the vast protential of the Scala language in Bioinformatics tasks. We present here some material of tutorial type.

BioScala (https://github.com/bioscala/bioscala) is available as a ScalaLab toolbox (file: bioScala.jar). Therefore, we can download bioScala.jar, and then install it, as any other ScalaLab toolbox. In addition, the BioJava toolbox, bioJava.jar, also available from ScalaLab's downloads, should be installed. The later is necessary since much functionality of BioScala is build upon BioJava (BioJava can also be used from ScalaLab, and a Scala based DSL for BioJava is in our future plans). After, performing these installation steps we can execute the following examples, that demonstrate some aspects of BioScala. The material in this page has been adapted from the documentation of BioScala.

The first step is to perform some imports:

import bio._

Then we can create a Sequence object

val dna = new DNA.Sequence("agctaacg")

Sequence is strongly typed. Creating a DNA sequence from an RNA String will give an error

val dna2 = new DNA.Sequence("agcuaacg")

You can transcribe DNA

val rna = dna.transcribe

rna: bio.RNA.Sequence = agcuaacg

Notice that the above is an RNA object now. Nucleotides are proper lists.

val l = RNA.A :: rna.toList

l: List[bio.RNA.Nucleotide] = List(a, a, g, c, u, a, a, c, g)

To translate nucleotides to amino acids:

bio.DNA.SequenceTranslation.translate(l)

res0: List[bio.Protein.AASymbol] = List(K, L, T)

Create a Sequence with ID

val seq = new DNA.Sequence("ID456", "agctaacg")
seq.id

Create a Sequence with ID and decsription

val seq = new DNA.Sequence("ID456", "My gene", "agctaacg")
seq.description

A Sequence, like in the real world, can have multiple ID's and descriptions.

```
import bio.attribute._
val seq2 = seq.attrAdd(Id("Pubmed:456"))
seq2.idList
val seq3 = seq2.attrAdd(Description("Another description"))
seq3.descriptionList
```

Note that Sequence is immutable. Every time you add an attribute a new copy gets created.

CodonSequence

The CodonSequence makes use of the Attribute list of the standard AminoAcidSequence object. When creating a CodonSequence, e.g.

```
val seq = new Protein.CodonSequence("ID356","Describe gene 356","agctgaatc")
seq.toString
seq.toDNA
```

it stored the DNA sequence as a CodonAttribute to the AminoAcid. When you want to fetch the codon sequence of Attribute with the third Amino Acid, you can query for the codon information

```
seq(2).getCodon
```

```
List(a, t , c)
```

and even directly from the CodonSequence

```
seq.getCodon(2)
```

Now we want to delete the middle codon:

```
var seq2 = seq.delete(1,1)
seq2.toDNA
```

We have another immutable CodonSequence object seq2, which contains a new sequence with matching amino acids and codons:

```
seq.toString
seq.toDNA
```

Computer Algebra

Introduction

Although ScalaLab is not a Computer Algebra System, we integrate within the embedded libraries, the symja (http://code.google.com/p/symja/) Java Computer Algebra system.

A top level menu choice, "Symbolic Algebra" aims to provide symbolic Algebra operations. This choice opens a GUI Window that allows the user to perform many Computer Algebra operations, e.g. to simplify expressions, to multiply/divide polynomials, factorizations, differentiations, integrations etc. Additionally, the user can easily plot 2-D and 3-D functions. Although, the operation of this window is currently independent from the rest ScalaLab components, we consider it very useful, since it is a way for the ScalaLab user to perform easily many Computer Algebra tasks.

In this page we present material that will facilitate the user to explore the potential of symja from within ScalaLab.

Using the symja Java Computer Algebra System from within ScalaLab

ScalaLab permits to perform Computer Algebra tasks with symja more conveniently and effectively than from the plain Java. For example we can run the following Scala code that uses symja to perform Computer Algebra tasks.

Examples demonstrating basic Symbolic Algebra tasks

The following Scala code uses symja to perform basic Computer Algebra tasks. You can run it, as any other ScalaLab script.

```
// Static initialization of the MathEclipse engine instead of null
// you can set a file name to overload the default initial
// rules. This step should be called only once at program setup:
F.initSymbols()
var util = new EvalUtilities()
 var buf = new StringBufferWriter()
 var input = "Expand[(AX^2+BX)^2]"
 var result = util.evaluate(input)
 OutputFormFactory.get().convert(buf, result)
 var output = buf.toString()
 println("Expanded form for " + input + " is " + output)
 // set some variable values
 input = "A=2;B=4"
 result = util.evaluate(input)
 buf = new StringBufferWriter()
```

```
input = "Expand[(A*X^2+B*X)^2]"
result = util.evaluate(input)
OutputFormFactory.get().convert(buf, result)
output = buf.toString()
println("Expanded form for " + input + " is " + output)

buf = new StringBufferWriter()
input = "Factor["+output+"]"
result = util.evaluate(input)
OutputFormFactory.get().convert(buf, result)
output = buf.toString()
println("Factored form for " + input + " is " + output)
```

And another example using symja from ScalaLab.

```
var br = new BigRational(0)
var vars = Array[String]("x")
var fac = new GenPolynomialRing[BigRational](br, vars.length, new
TermOrder(TermOrder.INVLEX), vars)
var eIntegrator = new ElementaryIntegration[BigRational](br)
var a = fac.parse("x^7 - 24 x^4 - 4 x^2 + 8 x - 8")
println("A: " + a.toString())
var d = fac.parse("x^8 + 6 x^6 + 12 x^4 + 8 x^2")
println("D: " + d.toString())
var gcd = a.gcd(d)
gcd = a.gcd(d)
var ret = eIntegrator.integrateHermite(a, d)
println("Result: " + ret(0) + " , " + ret(1))
println("-----")
a = fac.parse("10 x^2 - 63 x + 29")
println("A: " + a.toString())

d = fac.parse("x^3 - 11 x^2 + 40 x -48")
println("D: " + d.toString())

gcd = a.gcd(d)
println("GCD: " + gcd.toString())

ret = eIntegrator.integrateHermite(a, d)
println("Result: " + ret(0) + " , " + ret(1))
println("-----")
a = fac.parse("x+3")
println("A: " + a.toString());
d = fac.parse("x^2 - 3 x - 40")
println("D: " + d.toString())
gcd = a.gcd(d)
println("GCD: " + gcd.toString())
```

```
ret = eIntegrator.integrateHermite(a, d)
println("Result: " + ret(0) + " , " + ret(1))
println("-----")
a = fac.parse("10 x^2+12 x + 20")
println("A: " + a.toString())
d = fac.parse("x^3 - 8")
println("D: " + d.toString())
gcd = a.gcd(d)
println("GCD: " + gcd.toString())
ret = eIntegrator.integrateHermite(a, d)
println("Result: " + ret(0) + " , " + ret(1))
println("-----------------------------------------------------\n")
```

Linear Systems Solve

We provide examples on solving linear systems. Over-determined and under-determined cases. The *solve()* is the basic routine for solving linear systems A * x = b. That routine has many overloads that we list below.

```
def solve(A: ScalaSci.Mat,  b: ScalaSci.Mat) = {
   ScalaSci.Mat.solve(A, b)
}
 def solve(A: ScalaSci.EJML.Mat, b: ScalaSci.EJML.Mat) =
  A.solve(b)

 // solve with the CSparse
def solve(A: ScalaSci.Sparse,  b: Array[Double]) = {
   ScalaSci.Sparse.sparseSolve(A, b)
}
// solve with the MTJ Column Compressed Matrix
def solve(A: ScalaSci.CCMatrix, b: Array[Double]) = {
   ScalaSci.CCMatrix.BiCGSolve(A, b)
}
 def solve(A: Array[Array[Double]],  b: Array[Array[Double]]) = {
  LinearAlgebra.solve( A, b)
}
 def solve(A: ScalaSci.CommonMaths.Mat, b: ScalaSci.CommonMaths.Mat): ScalaSci.CommonMaths.Mat =
  new ScalaSci.CommonMaths.Mat(solve(A.v, b.v) )


def solve(A: ScalaSci.JBLAS.Mat, B: ScalaSci.JBLAS.Mat) =  {
  new ScalaSci.JBLAS.Mat(org.jblas.Solve.solve(A.dm, B.dm))
}
def solve(A: RichDoubleDoubleArray,  b: RichDoubleDoubleArray) = {
  new RichDoubleDoubleArray(LinearAlgebra.solve( A.getv, b.getv))
}
 def solve(A: RichDoubleDoubleArray,  b: RichDoubleArray): RichDoubleArray = {
   var N = b.length
```

```
    var bb = new RichDoubleDoubleArray(N,1)
    for (k<-0 until N)
      bb(k, 0) = b(k)
    var sol = solve(A, bb)
    // convert the solution to a RichDoubleArray
    var x = new Array[Double](sol.numRows)
    for (k<-0 until sol.numRows)
      x(k) = sol(k, 0)
    new RichDoubleArray(x)

}
  def solve(A:ScalaSci.MTJ.Mat, B: ScalaSci.MTJ.Mat, X: ScalaSci.MTJ.Mat) =
      A.solve(B, X)
```

There are also other routines for solving linear systems (of course one can use the native Java interfaces of the ScalaLab libraries). We demonstrate some solving routines in the following examples.

Examples

```
val tol = 0.000001  // tolerance  on error
// a two-dimensional double array
var  A = AAD("2.11 -4.21 0.921; 4.01 10.2 -1.12; 1.09 0.987 0.832")
// a 1-d double array
var  b = AD("2.01, -3.09, 4.21")
// solve with LAPACK
var x = la_LUSolve(A, b)
var OK_la_LU = if  (max(A*x-b) < tol) true else false
// JAMA based solver requires  b as an Array[Array[Double]] with the right-hand sides of the equations as columns
var  bb = AAD("2.01;-3.09; 4.21")
var xjama = jama_LUSolve(A, bb)
var OK_la_JAMA = if  (max(max(A*xjama-bb)) < tol) true else false
// standard solve()  method
var xsolve  = solve(A,b)
var OK_solve = if (max(A*xsolve-b)<tol) true else false
```

Demonstration of exact, overdetermined and underdetermined cases

The following example demonstrates both the N equations by N unknowns case and the overdetermined and underdetermined ones.

```
//  define N equations by N uknowns system
var A = AAD("3.5 5.6 -2; 4.5 6.7 9.4; 2.3 -2.3 9.4")
var b = AAD("-0.2; 0.45; -3.4")
var x= solve(A, b)   // solve the linear  system
A*x-b   // verify that is zero
var x2 = A \ b    // we can solve it with the operator '\'
A*x2- b  // verify that is zero
// test DGELS
var lssol = DGELS(A, b)  // solve with the LAPACK routine
A*lssol-b  // verify that is zero
// define an over determined system, i.e. number of equations > number of uknowns
var ARowsMoreThanCols = AAD("3.5 5.6 -2; 4.5 6.7 9.4; 2.3 -2.3 9.4; 8.1 2.3 -0.2")
var bRowsMoreThanCols = AAD("-0.2; 0.45; -3.4; -0.3")
// solve overdetermined system with the LAPACK routine
var lssolOverDetermined = DGELS(ARowsMoreThanCols, bRowsMoreThanCols)
/* MATLAB
  ARowsMoreThanCols = [3.5 5.6 -2; 4.5 6.7 9.4; 2.3 -2.3 9.4; 8.1 2.3 -0.2];
  bRowsMoreThanCols = [-0.2; 0.45; -3.4; -0.3];
  lssolOverDetermined = ARowsMoreThanCols \ bRowsMoreThanCols
 */
// define an under determined system, i.e. number of equations < number of unknows
 var AColsMoreThanRows= AAD("13.5  5.6  9.3 -2; 3.5 3.7 9.4 -0.7; 2.3 -2.3 0.9 9.4")
var bColsMoreThanRows= AAD("-0.4; 0.5; -3.4")
// solve with the LAPACK routine
var lssolUnderDetermined = DGELS(AColsMoreThanRows, bColsMoreThanRows)
AColsMoreThanRows*lssolUnderDetermined
/* MATLAB
  AColsMoreThanRows = [13.5  5.6  9.3 -2; 3.5 3.7 9.4 -0.7; 2.3 -2.3 0.9 9.4];
  bColsMoreThanRows = [-0.4; 0.5; -3.4; 9.7];
  lssolUnderDetermined = AColsMoreThanRows  \ bColsMoreThanRows
 */
```

Example for the MATLAB-like backslash operator

The following example uses the backslash MATLAB-like operator to demonstrate solving linear systems using various libraries

```
// testing \ (backslash) operator for solving linear systems
// RichDoubleDoubleArray
var ARDDA = AAD("3.4 4.5; -1.2 5.6")
var bRDDA = AAD("4.3; 9.2")
var xRDDA = ARDDA \  bRDDA
var shouldBeZeroRDDA = ARDDA*xRDDA - bRDDA
// JAMA Matrix
var AJamaMat = ScalaSci.StaticMaths.M0("3.4 4.5; -1.2 5.6")
var bJamaMat = ScalaSci.StaticMaths.M0("4.3; 9.2")
var xJamaMat = AJamaMat \ bJamaMat
var shouldBeZeroJamaMat = AJamaMat*xJamaMat-bJamaMat
// EJML Matrix
var AEJMLMat = ScalaSci.EJML.StaticMathsEJML.M0("3.4 4.5; -1.2 5.6")
var bEJMLMat = ScalaSci.EJML.StaticMathsEJML.M0("4.3; 9.2")
var xEJMLMat = AEJMLMat \ bEJMLMat
var shouldBeZeroEJMLMat = AEJMLMat*xEJMLMat-bEJMLMat
// MTJ Matrix
var AMTJMat = ScalaSci.MTJ.StaticMathsMTJ.M0("3.4 4.5; -1.2 5.6")
var bMTJMat = ScalaSci.MTJ.StaticMathsMTJ.M0("4.3 ; 9.2")
var xMTJMat = AMTJMat \ bMTJMat
var shouldBeZeroMTJMat = AMTJMat*xMTJMat-bMTJMat
// Apache Commons  Matrix
var AApacheCommonsMat = ScalaSci.CommonMaths.StaticMathsCommonMaths.M0("3.4 4.5;
-1.2 5.6")
var bApacheCommonsMat = ScalaSci.CommonMaths.StaticMathsCommonMaths.M0("4.3 ; 9.2")
var xCommonMathsMat = AApacheCommonsMat \ bApacheCommonsMat
var shouldBeZeroApacheCommonsMat = AApacheCommonsMat*xCommonMathsMat-
bApacheCommonsMat
// JBLAS  Matrix
var AJBLASMat = ScalaSci.JBLAS.StaticMathsJBLAS.M0("3.4 4.5; -1.2 5.6")
```

```
var cpAJBLASMat = new ScalaSci.JBLAS.Mat(AJBLASMat.getv)  // copy since Native BLAS overwrittes original
var bJBLASMat = ScalaSci.JBLAS.StaticMathsJBLAS.M0("4.3 ; 9.2")
var cpbJBLASMat = new ScalaSci.JBLAS.Mat(bJBLASMat.getv)  // copy since Native BLAS overwrittes original
var xJBLASMat = AJBLASMat \ bJBLASMat
var shouldBeZeroJBLASMat = cpAJBLASMat*xJBLASMat   -cpbJBLASMat
// sparse solve
var Asparse = SparseFromDoubleArray(ARDDA.getv)
var b = Array(4.3, 9.2)
var sol = Asparse  \ b  // solve the sparse system
```

Complex Numbers

Introduction

ScalaLab intents to support complex numbers and complex matrices as integrated elements of the environment.

The *ScalaSci.Complex* class implements much functionality for complex numbers.

We can create a complex number, nicely as e.g.

*var x = 3.4 +0.23*i*

However, *i* should not be declared as a variable.

Representation of a Complex number, i.e. a number which has both a real and imaginary part.

I

public static final Complex **I**

The square root of -1. A number representing "0.0 + 1.0i"

ONE

public static final Complex **ONE**

A complex number representing "1.0 + 0.0i"

ZERO

public static final Complex **ZERO**

A complex number representing "0.0 + 0.0i"

Constructors

new Complex(real: Double)

Create a complex number given only the real part, e.g.

var x = new Complex(2.3)

*new **Complex**(real: Double, imaginary: Double)*

Create a complex number given the real and imaginary parts, e.g.

var x = new Complex(3.4, -2.3)

Methods

+, -, *, /

Perform the usual addition, subtraction, multiplication and division operations

def abs(): Double

Return the absolute value of this complex number.

def conjugate(): Complex

def conj(): Complex

Return the conjugate of this complex number. The conjugate of a + bi is a - bi.

def reciprocal(): Complex

Returns the multiplicative inverse of this element.

def getReal(): Double

Access the real part.

def getIm(): Double

Access the imaginary part.

def acos(): Complex

Compute the *inverse cosine* of this complex number. Implements the formula:

acos(z) = -i* (log(z + i* (sqrt(1 – z^2))))

e.g.

*var x = 3.4 - 6.7 * i*

var shouldBeZero = acos(x) - (-i(log(x+i*(sqrt(1-x*x)))))*

def asin(): Complex

Compute the *inverse sine* of this complex number. Implements the formula:

asin(z) = -i * (log(sqrt(1 - z^2) + i*z))

e.g.

*var x = 3.4 - 6.7 * i*

var shouldBeZero = asin(x) - (-i(log(sqrt(1-x*x)+i*x)))*

def atan(): Complex

Compute the *inverse tangent* of this complex number. Implements the formula:

atan(z) = (i/2) log((i + z)/(i - z))

e.g.

*var x = 3.4 - 6.7 * i*

*var shouldBeZero = atan(x)-(i/2)*log((i+x)/(i-x))*

def cos(): Complex

Compute the *cosine* of this complex number. Implements the formula:

cos(a + bi) = cos(a)cosh(b) - sin(a)sinh(b)i

*var x = 3.4 - 6.7 * i*

*var shouldBeZero = cos(x)-cos(3.4)*cosh(-6.7) + sin(3.4)*sinh(-6.7)*i*

def cosh(): Complex

Compute the *hyperbolic cosine* of this complex number. Implements the formula:

cosh(a + bi) = cosh(a)cos(b) + sinh(a)sin(b)i}

*var x = 3.4 - 6.7 * i*

*var shouldBeZero = cosh(x)-cosh(3.4)*cos(-6.7) - sinh(3.4)*sin(-6.7)*i*

def exp(): Complex

Compute the *exponential function* of this complex number. Implements the formula:

exp(a + bi) = exp(a)cos(b) + exp(a)sin(b)i

*var x = 3.4 - 6.7 * i*

*var shouldBeZero = exp(x)-exp(3.4)*cos(-6.7) - exp(3.4)*sin(-6.7)*i*

def log(): Complex

Compute the *natural logarithm* of this complex number. Implements the formula:

log(a + bi) = ln(|a + bi|) + arg(a + bi)i

def pow(x: Double): Complex

Returns of value of this complex number raised to the power of x. Implements the formula:

$y^x = \exp(x \cdot \log(y))$

def sin(): Complex

Compute the *sine* of this complex number. Implements the formula:

sin(a + bi) = sin(a)cosh(b) - cos(a)sinh(b)i

var x = 3.4 - 6.7 * i

var shouldBeZero = sin(x)-sin(3.4)*cosh(-6.7) + cos(3.4)*sinh(-6.7)*i

def sinh(): Complex

Compute the *hyperbolic sine* of this complex number. Implements the formula:

sinh(a + bi) = sinh(a)cos(b) + cosh(a)sin(b)i

var x = 3.4 - 6.7 * i

var shouldBeZero = sinh(x)-sinh(3.4)*cos(-6.7) - cosh(3.4)*sin(-6.7)*i

def sqrt(): Complex

Compute the square root of this complex number.

def sqrt1z(): Complex

Compute the square root of 1 - this^2 for this complex number. Computes the result directly as sqrt(ONE.subtract(z.multiply(z))).

var x = 3.4 - 6.7 * i

var shouldBeZero = x.sqrt1z - sqrt(1-x*x)

def tan(): Complex

Compute the *tangent* of this complex number. Implements the formula:

tan(a + bi) = sin(2a)/(cos(2a)+cosh(2b)) + [sinh(2b)/(cos(2a)+cosh(2b))]i

def tanh(): Complex

Compute the *hyperbolic tangent* of this complex number. Implements the formula:

tan(a + bi) = sinh(2a)/(cosh(2a)+cos(2b)) + [sin(2b)/(cosh(2a)+cos(2b))]i

def getArgument(): Double

Compute the argument of this complex number. The argument is the angle phi between the positive real axis and the point representing this number in the complex plane. The value returned is between -PI (not inclusive) and PI (inclusive), with negative values returned for numbers with negative imaginary parts.

def nthRoot(): List[Complex]

Computes the n-th roots of this complex number. The nth roots are defined by the formula:

$z_k = abs^{1/n} (cos(phi + 2\pi k/n) + i (sin(phi + 2\pi k/n))$

for *k=0, 1, ..., n-1*, where abs and phi are respectively the modulus and argument of this complex number.

References:

[1] Stephen L. Campbell, Jean-Philippe Chancelier, Ramine Nikoukhah, *Modeling and Simulation in Scilab/Scicos*, Springer, 2006

[2] Cay Horstmann, Gary Cornell, *Core Java 2*, Vol I Fundamentals, Vol II - Advanced Techniques. Sun Microsystems Press, 8th edition, 2008

[3] Norman Chonacky, David Winch, *3Ms for Instruction: Reviews of Maple, Mathematica and MATLAB*, IEEE Computing in Science and Engineering (CISE), May/June 2005, Part I, pp. 7-13

[4] Norman Chonacky, David Winch, *3Ms for Instruction: Reviews of Maple, Mathematica and MATLAB*, IEEE Computing in Science and Engineering (CISE), July/August 2005, Part II, pp. 14-23

[5] John W. Eaton, "*GNU Octave Manual*, Network Theory Ltd, 2002

[6] Desmond J. Higham, Nicholas J. Higham, *MATLAB Guide*, Second Edition, SIAM Computational Mathematics, 2005

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

[8] Papadimitriou S, Terzidis K., *jLab: Integrating a scripting interpreter with Java technology for flexible and efficient scientific computation*, Computer Languages, Systems & Structures (2008), Elsevier, vol. 35, 2009, pp. 217–240

[9] Dierk Konig, Andrew Glover, Paul King, Guillaume Laforge, Jon Skeet, *Groovy In Action*, Manning Publications, 2007

[10] Simon Haykin, "Neural Networ*ks* and Learning Machines", Third Edition, Pearson Education, *2009*

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

[12] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical Recipes in C++, The Art of Scientific Computing, Second Edition*, Cambridge University Press, Third-edition, 2007

[13] Alfred Aho, Monica S. Lam, Ravi Sethi, Jefrey D. Ullman, Compilers, Principles, Techniques, & Tools, Second Edition, Addison-Wesley, 2007

[14] Stergios Papadimitriou, Konstantinos Terzidis, Seferina Mavroudi, Spiridon Likothanasis, "Scientific Scripting for the

Java Platform with jLab", IEEE Computing in Science and Engineering (CISE), July/August 2009, Vol. 11, No 4, pp. 50-60

[15] Martin Odersky, Lex Spoon, Bill Venners, Programming in Scala, Artima eds, 2008

[16] Dean Wampler & Alex Payne, *Programming Scala*, O'Reily, 2009

[17] Venkat Subramaniam, *Programming Scala – Tackle Multicore Complexity on the Java Virtual Machine*, Pragmatic Bookself 2009

[18] Thomas Wurthinger, Christaian Wimmer, Hanspeter Mossenblock, "Array Bounds Check Elimiation for the Java Hotspot Client Compiler", PPPJ 2007, September 5-7, 2007, Lijboa Portugal, ACM

[19] Stergios Papadimitriou, Konstantinos Terzidis, Seferina Mavroudi, Spiridon 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), in print

Alphabetical Index