Scientific Computing with GroovyLab at the Java Platform

Table of Contents

The GroovyLab environment....7
- Introduction....7
- The architecture of the GroovyLab environment....9
- Working with the Groovy Shell Pane....13

Installing Toolboxes....14
The import buffering mechanism....15
- Binding....18
- Operator Overloading....19

Tutorial on the Vec class....20
Tutorial on Matrix class....23
- Matrix Construction....24
- Matrix Indexing/Setting operations....26
- Matrix subrange operations....27
- Operators....32
- More convenient access/assignment operations....32
- Static operations....33

Numerical Analysis with GroovyLab....34
GroovySci Examples....35
- Henon Chaotic Map....35
- Baker map....36
- Ikeda Chaotic Map....37
- Demonstrating surface plots....37
- A cloud plot demo....39
- Perform some plots that test the VISAD interface, requires installation of Java 3D....39
- A Continuous Wavelet Transform example....40
- Plotting using a MATLAB-like interface to the JFreeChart library....41
- Plotting Demonstrations....43
- And other plotting Demonstrations....44
- Various sine functions on the same plot....44
- Using the Groovy's SwingBuilder to specify signal parameters and process a signal....45
- Some simple signals with induced noise....46
- Functional Plotting and plotting functions specified as expressions....46
  - One dimensional function plotting....46
- Object – Oriented Plotting....47
  - Design of a Plot Facilitator Object (this code exists in GroovyLab source)....47
  - Testing the Plot Controller Object....50
- The jzy3D for scientific plotting in GroovyLab....51
  - 3-D Plotting of a function....51
  - A Matlab like interface for plotting with jzy3D....53
  - Executing a Java example (with F9 keystroke)....53
- Exploiting Java Libraries from GroovyLab....71
  - EJML Example....73

Fast Computations with the JBLAS library....74
JBLAS operations supported with the Matrix class....74
Matrix multiplication with JBLAS....78
Switching of the GroovySci Matrix class to use JBLAS....79
Eigendecomposition with JBLAS....80
Solution of Linear Systems with JBLAS....82
FFT With Numerical Recipes Routines....83
Extending the functionality of the GroovySci Matrix using EJML Routines....87
Transforming GroovyLab scripts to standalone applications....90
Wavelets with GroovyLab....91
Sparse Matrix class based on MTJ Sparse Matrices....93
Sparse Matrix support with the Sparse Class....95
Compatibility with Matlab .mat file format....98
Using JTransforms from GroovyLab....100
References:....102
Apache Common Maths....103
Descriptive Statisticstics....103
Simple Regression....105
Multiple linear regression....107
Rank transformations....109
Covariance and correlation....109
Statistical tests....112
Numerical Analysis....117
4.1 Overview....117
Error handling....117
Root-finding....118
Interpolation....121
Integration....122
Polynomials....123
Differentiation....123
Data Generation....126
Random numbers....126
Random Vectors....127
Random Strings....128
Random permutations, combinations, sampling....129
Generating data 'like' an input file....129
PRNG Pluggability....130
Special Functions....133
5.1 Overview....133
5.2 Erf functions....133
5.3 Gamma functions....133
Gamma....133
Log Gamma....133
Regularized Gamma....134
5.4 Beta functions....134
Log Beta....134
Regularized Beta....135
Linear Algebra....136
Overview....136
Real matrices....136
Real vectors....137
Solving linear systems....137
Eigenvalues/eigenvectors and singular values/singular vectors....138

Non-real fields (complex, fractions ...)....138
Utilities....139
Double array utilities....139
int/double hash map....139
Continued Fractions....139
Binomial coefficients, factorials, Stirling numbers and other common math functions....140
Fast mathematical functions....140
Miscellaneous....141
Complex Numbers....142
Probability Distributions....145
8.1 Overview....145
8.2 Distribution Framework....145
8.3 User Defined Distributions....145
Fractions....147
9.1 Overview....147
9.2 Fraction Numbers....147
9.3 Fraction Formatting and Parsing....147
Transforms....149
org.apache.commons.math3.transform
Class FastFourierTransformer....149
FastFourierTransformer....150
transformInPlace....150
transform....150
transform....151
transform....151
mdfft....152
org.apache.commons.math3.transform
Class FastCosineTransformer....152
FastCosineTransformer....153
transform....153
transform....154
fct....154
org.apache.commons.math3.transform
Class FastSineTransformer....155
FastSineTransformer....156
transform....156
transform....157
fst....157
org.apache.commons.math3.transform
Class FastHadamardTransformer....158
FastHadamardTransformer....159
transform....159
transform....159
transform....160
fht....160
Short Table of manual calculation for N=8....160
How it works....161
Visually....161
fht....162
Geometry....163
11.1 Overview....163
11.2 Euclidean spaces....163
11.3 Binary Space Partitioning....164

Optimization........165
12.1 Overview........165
12.2 Univariate Functions........165
12.3 Linear Programming........166
12.4 Direct Methods........166
12.5 General Case........167
12.6 Curve Fitting........170
Ordinary Differential Equations Integration........172
13.1 Overview........172
13.2 Continuous Output........173
13.3 Discrete Events Handling........174
13.4 Available Integrators........175
13.5 Derivatives........175
Genetic Algorithms........178
14.1 Overview........178
14.2 GA Framework........178
14.3 Implementation........178
17.1 Overview........180
17.2 General Case........180
7.3 Special Cases........181

Preface

The GroovyLab environment builds upon Groovy, a new language for the Java Virtual Machine with full Java interoperability. The book describes how Groovy can be extended with MATLAB-like constructs in order to implement a powerful compiled mathematical scripting framework. The reader should have programming experience preferably with Java or even better with Groovy. However, for GroovyLab programming ***advanced programming techniques are not required, although they can be used****. One of the main objectives of GroovyLab 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 Groovy language on which GroovyLab is build upon.*

GroovyLab 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 GroovyLab explores the potential of Java's Swing library in order to facilitate the work of the scientist. The scripting language of GroovyLab is **GroovySci** which offers MATLAB-like high-level mathematical operators by exploiting the flexibility of the Groovy language for building easily new syntax. In addition, the user of GroovyLab can mix easily and Java code that can be compiled with an internal Java 6 compiler. An external **javac** compiler whenever is available can also be used to compile Java code, and can be called directly from GroovyLab, but the main point is that an installation of the Java JDK is not required to compile Java code.*

*The GroovyLab **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 Groovy and Java code. Also, classpath related operations can be managed from the GroovyLab explorer.*

*An extension of Groovy with MATLAB-like constructs, called **GroovySci** is the language of GroovyLab. GroovySci is effective both for writing small scripts and for developing large production level applications. The book describes the core scientific classes of GroovySci, and presents examples of their application.*

Since Groovy has a new static typing mode, GroovySci code can run 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, GroovyLab can run heavy numerical tasks with speed competitive to the traditional C/C++ or Fortran compiled code.

The GroovyLab editor is based on the **rsyntaxarea** *editor (*http://fifesoft.com/rsyntaxtextarea/*) 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.*

The GroovyLab 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 internal Java compiler, compile of Groovy sources with the Groovy compiler, single-step GroovySci code and text coloring.

By exploiting the flexibility and extensibility of the Groovy language we also present the framework for the utilization of Java scientific libraries within the GroovyLab

*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 and **EJML** (Efficient Java Matrix Library). Java remains a good and effective solution for simple tasks as the development of scientific software. Thus, the philosophy of GroovyLab 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. Therefore, a Java 6 compiler is integrated within the core of GroovyLab.*

In GroovyLab, 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 GroovyLab.

The GroovyLab environment

Introduction

Numerical computation applications benefit from *interactive systems for matrix computation,* which facilitate the rapid scientific experimentation. With these scripting systems substantial analysis can be performed by entering stepwise commands and thereby obtaining results. Experimentation is encouraged and the tedious ``compile-link-execute'' cycle of standard programming languages is eliminated. Well known examples of such systems include commercial products 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 GroovyLab environment [16]. Although GroovyLab presents a MATLAB-like style of working, it differs from the forementioned in that it compiles the scripts for the JVM. GroovyLab utilizes and expands the powerful Groovy dynamic language [0].

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 [15] 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. GroovyLab exploits both the flexibility of Groovy and the robust and feature rich Java platform in order to offer an open MATLAB-like system to the scientist. The user of GroovyLab can work both with the MATLAB-like script oriented way or with the Java/Groovy like way of building stand-alone applications. Also, the GroovyLab 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 GroovyLab. 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 Groovy provides. Many features of Java's Swing assist the work of the user, for example by providing extensive help and superb display functionality.

The architecture of the GroovyLab environment

GroovyLab builds upon the technology stack developed over many years for the Java Virtual Machine. 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 GroovyLab with either a *client* or a *server* **JVM** and sometimes the server JVM has significant improvement in performance for heavy computational tasks.

b. The Java Compiler

GroovyLab integrates the Java 6 compiler within its executable. Thus, the user can easily compile and run Java classes, avoiding the inconvenience of installing and handling an external Java compiler.

c. The Groovy Compiler and Run Time Libraries

GroovyLab integrates a recent version of the whole Groovy system. Thus, a seperate installation of Groovy is unecessary.

d.The Groovy Shell

This component is the core of GroovyLab and the one that implements the MATLAB-like sense of the system. It is an advanced interpreter that maintains state between successive script executions.

e. The integrated Java Libraries

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

f. External class libraries

Since GroovyLab runs upon the Java Virtual Machine any class file can be executed as long as it is

placed on the GroovyLab classpath. GroovyLab keeps an adaptable environment variable *GroovySciClassPath* for implementing the notion of classpath. That variable controls the classpath of the Groovy Shell.

However, special support is offered for conveniently loading .jar files that pack Java libraries, since it is especially important to utilize effectively toolboxes of Java code.

g. Application Level Wizards

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

h. The Computer Algebra Subsystem

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

i. The Imports Wizard

GroovyLab uses a large number of libraries. This wizard allows to perform easily the imports required for each library by injecting the proper import commands.

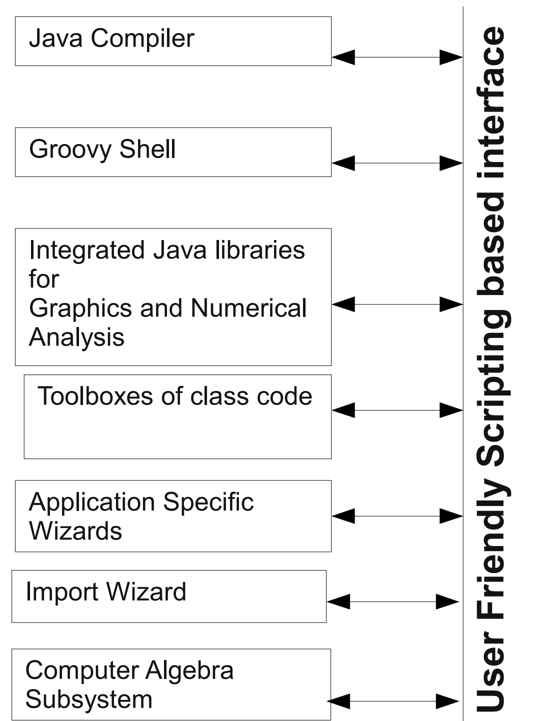


Figure 1 The Architecture of the GroovyLab

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

a. The *GroovyLab Interpreter Pane* which is based on the **RsyntaxTextArea** (http://fifesoft.com/rsyntaxtextarea/) 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. This interface is the preferred interface for writing scripts. However, an additional console based interface exists, the *GroovyLab console.*

b. The *GroovyLab console* receives commands and scripts from the user. It is important to note that both the *GroovyLab Interpreter Pane* and the *GroovyLab Console* share the same Groovy Shell.

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

d. The *GroovyLab editor*, is a simple text editor based on the **rsyntaxarea** component, that however assists by offering special operations such as compiling and running Java/Groovy 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 Groovy Compiler that produces when the Groovy Shell feeds to it the script for compilation.

f. The *variable workspace* window that displays the globally accessible variables and their types.

g. The *import wizard* accomplishes useful tasks aimed at facilitating import handling, as for example the import buffering described below.

Working with the Groovy Shell Pane

The GroovyLab environment allows to easily execute GroovyLab code. We describe here the main points of working with the Groovy Shell Pane editor.

Perhaps the most important keystroke is the **F6** that permits to execute either:

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

Similarly with the **F6** keystroke works the **F8** keystroke, with the difference that it permits the execution of the script in a separate thread. Therefore, GroovyLab is not blocked, while computation is executed

For example, if we have the line of code:

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

Then to execute the whole line, we can press F6 with the cursor anywhere in the line. To execute only the command concerning variable *beta* 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 binded variables by the GroovyShell, displayed at the ***table of binding variables*** are also understable as completions, e.g. if we have defined the variable: *myIntVarWithLargeName = 8*, and then press *myInt* and *Ctrl-SPACE* the code completion will complete the rest name. We can update at any time the table of binding variables by pressing **F12.**

The **Ctrl-SPACE** keystroke activates the code completion "Provider" of the RsyntaxTextArea component. The RSyntaxArea editor supports **code completion (using CONTROL-SPACE) in two modes**, that are switchable with a menu option of the *Completion* menu. These modes are:

1. ***Global completion mode***. This mode is useful to remind the global GroovyLab methods, e.g. the many overloaded versions of *plot()*. The global completion list can be extended, using library routines, that are detected using Java reflection. These libraries are available at the *Completion* menu.

2. ***Groovy completion mode***. This mode works by exploiting the information acquired from *Java Reflection.* It is very useful to perform *field and method* completions.

The **F1** key is also useful, since it collects information from the basic GroovyLab 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 GroovyLab. We can select "fft" and press F1. Then a list is displayed with the GroovyLab classes that provide *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 **F9** keystroke concerns the compilation and execution of Java code with the internal Java compiler of GroovyLab and is thus irrelevant when we work with GroovySci scripts.

The **F10** keystroke on a selected identifier, displays information for the class of the selected identifier by using the *Groovy Object Browser*. For example, let:

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

selecting "*jf*" and pressing F10, we have a Groovy Object Browser based presentation of the contents of the JFrame class.

Installing Toolboxes

Installation of .jar toolboxes is very important and allows convenient access to Java scientific libraries. In GroovyLab we can easily install .jar toolboxes that remain installed for later working sessions. We illustrate installation of a toolbox by means of an example.

Example

Suppose that we want to install the jWave.jar toolbox. We proceed by opening the GroovySci Toolboxes tab. A frame titled GroovySci toolboxes is displayed that allows convenient installation of toolboxes.

The Specify toolboxes button allows to browse the file system and to specify our toolbox. The Append to classpath button appends the toolbox to the classpath of the GroovyShell

without attempting to scan its classes. It is sufficient for our purpose of using the classes of the toolbox, since in order to use a toolbox the only requirement is to be accessible from the GroovyShell's classpath. The Import toolboxes button in addition scans the toolbox classes and if the Retrieve also methods checkbox is checked, acquires information about the toolbox classes with Java's reflection. That information is displayed graphically by means of a Swing's JTree.

Let now proceed in installing our **jWave.jar** toolbox. We follow these steps:

1. With Specify toolboxes we specify the jWave.jar file.
2. We can append the toolbox to the classpath of a fresh GroovyShell that however keeps the binding of the variables of the previous one, with Append to classpath button. Alternatively, we can use the Import toolboxes button that attempts to scan the toolbox classes, presenting with useful information about toolbox classes, and methods if the Retrieve also methods checkbox is checked.

After installing the toolbox we can execute the example code for the jWave toolbox (see the Wiki page).

Now we can exit GroovyLab. A file **GroovyLabUserPaths.txt** is created on disk, that contains the line:

/home/sp/Downloads/jWave.jar

This line specifies to GroovyLab that on startup should init the GroovyShell's classloader to include also the jWave.jar toolbox. The next time we run GroovyLab we can use jWave.jar without installation.

To remove the **jWave.jar** toolbox we can click on its node under the *GroovyShell Classpath* node. Then the jWave.jar is not available at the next GroovyLab session.

The import buffering mechanism

Characteristics of Groovy explored in GroovyLab

The GroovyShell lacks a feature of keeping the imports from the previous statements. This

can be a rather inconvenient in many cases, since we must issue manually the same import statements, before executing commands depending on them. For that reason, we implemented in GroovyLab a simple import buffering, that we think can facilitate the user's work.

GroovyLab prepends at the user's scripts some basic imports that extend the functionality, by performing a lot of import and import static statements, and allowing for example to use commands as *figure()*, *plot()* etc. In order to allow specialized import statements to be kept during a user's session, the user can **buffer** these import statements for that particular working session only. That can be accomplished by **selecting the relevant import statements** and then using **Buffer selected Imports** option from the **Imports** menu.

As an example suppose that we have the following script

```
// SECTION 1:  import section
import static com.nr.test.NRTestUtil.maxel
import static com.nr.test.NRTestUtil.vecsub
import static java.lang.Math.cos
import com.nr.interp.RBF_gauss
import com.nr.interp.RBF_interp
import com.nr.interp.RBF_inversemultiquadric
import com.nr.interp.RBF_multiquadric
import com.nr.interp.RBF_thinplate
import com.nr.ran.Ran
// SECTION 2: import independent commands section
  NPTS=100; NDIM=2; N=10; M=10;
  sbeps=0.05
  pts =new double[NPTS][NDIM]
  y = new double[NPTS]
  actual = new double[M]
  estim = new double[M]
  ppt = new double[2]
  globalflag=false

 // SECTION 3:  import dependent command section
```

```
// Test RBF_interp
 myran = new Ran(17)
 pt = new double[M][2]
 for (i=0;i<M;i++) {
  pt[i][0]=(double)(N)*myran.doub()
  pt[i][1]=(double)(N)*myran.doub()
  actual[i]=cos(pt[i][0]/20.0)*cos(pt[i][1]/20.0)
}
 for (i=0;i<N;i++) {
  for (j=0;j<N;j++) {
   k=N*i+j
   pts[k][0]=(double)(j)
   pts[k][1]=(double)(i)
   y[k]=cos(pts[k][0]/20.0)*cos(pts[k][1]/20.0)
  }
}
 println("Testing RBF_interp with multiquadric function")
 scale=3.0
 multiquadric = new RBF_multiquadric(scale)
 myRBFmqf = new RBF_interp(pts,y,multiquadric,false)

 for (i=0;i<M;i++) {
  ppt[0]=pt[i][0]
  ppt[1]=pt[i][1]
  estim[i]=myRBFmqf.interp(ppt)
}
```

In order to execute first the import independent commands (i.e. SECTION 2), and after them in a separate script, the import dependent commands (i.e. SECTION 3) the user should *select the imports* and buffer them by using from the "*Imports*" menu the "*Buffer selected imports*" option. Issuing multiple "*Buffer selected imports*" commands, accumulate all the import statements, unless we clear the import buffer.

As we expect the script as a whole executes correctly. However, if we do not buffer the imports and try to execute statements that depend on the imports (e.g. myran = new Ran(17)), without prepending the code with them, we fail. However, if we buffer the necessary imports, we can execute our script conveniently **command-by-command**.

Binding

Binding of variables with GroovyShell is only for **undeclared data variables**. It is important that although GroovyShell does not keep objects and classes, it keeps *closure* objects which behave much like *functions* in functional programming.

For example, we can define a *cube* closure as:

*cube = { x -> x*x*x }*

Thereafter, we can apply it:

x = 10

y = cube(x)

y

We can also apply a closure on a Matrix:

add1000 = { x -> 1000+x}

a = rand(3,3)

a.eachValue(add1000)

In dynamic languages, as e.g. Groovy, the building of high-level mathematical operators is based on the *Metaobject protocol* that forms the base for the implementation of the dynamic method invocation [9]. This protocol is the base on which the flexibility of the language is implemented. Besides, the metaobject protocol permits Groovy to keep the Java syntax intact. In dynamic languages, usually we can inject methods into a class by adding methods to its *MetaClass.* These methods are available globally. In Groovy, with this mechanism we can add methods, properties, constructors and static methods. We can

inject new methods both to Groovy objects and to Java ones.

For example the *Number* class of the standard Groovy's library does not implement an addition with an array of values. Even though, we can easily exploit the potential of the Metaobject protocol to define this method, as illustrated with the following Groovy code:

```
//  define the operation: Number + double [],
// at the MetaClass of the Number's class
Number.metaClass.plus =
   {
//  the double [] m array denotes the input parameter
  double []  m ->

// call a Java routine for the operation
           res =
groovySci.math.LinearAlgebra.LinearAlgebra.plus( delegate,  m)

        res
}
```

The keyword *delegate* refers to the current object, i.e. the *Number* object. Also, since Groovy's bytecode is somewhat slow for number crunching, GroovyLab intermixes Java code to attain good performance.

Operator Overloading

In Groovy each operator has a standard mapping to methods. From Java only these methods can be used, but from Groovy we can use either the operators or their corresponding methods.

We can provide operators for our classes by adding the mapping methods, like *plus()* for the addition operator “+”. Although dynamic, Groovy achieves type safety with strongly typed design. Unlike untyped languages, Groovy doesn't allow to treat an object of one type as an instance of a different type without a well-defined conversion.

Tutorial on the Vec class

The **Vec** class implements one dimensional vectors, in a rather flexible way. Its base representation is a one-dimensional **double []** array type.

One way to construct a Vec is by passing the elements in a string as:

v = V(" 7, 8.7, -56.7") // comma separated

or

v = V(" 7 8.7 -56.7") // space separated

Even better is to use the constructor, *Vec(double ..args),* i.e. to pass a variable number of arguments, that become the vector elements. In this way we can use any expression as a vector element. For example:

elem1 = 4.5; elem2 = -6

*x = V(elem1, 4*sin(elem1), 5.6, elem2-elem1)*

We can construct an empty vector as:

v = new Vec(100) // an 100 element vector initialized to zeros

We can construct it also from a double [] array, either by copying it, or simply having the double array as a reference (faster):

da = [7.8, -5.6, 4.5] as double[]

vda = new Vec(da) // copy

vrda = new Vec(da, true) // reference

vda[2] = 22 // da[2] is not changed

vrda[2] = 22 // now da[2] is changed

We can get the double [] array representation of a Vec with:

v = V(7, 8.7, -56.7)

dv = v.getv()

The length of the vector is returned with *length()* or *size():*

vl = v.length()

vs = v.size()

We can apply a ***Closure*** to each vector's element with *map()* or *eachValue():*

v = vlinspace(0, 10, 10000) // linearly spaced vector of 10000 values between 0 and 10

vc = v.copy() // keep a copy of the vector

*sf = { x -> sin(0.34*x) } // define a closure*

vy = v.map(sf)   // map the closure returning the vector vy

plot(v, vy, Color.RED)   // plot the results

hold(true)

vc.i_map(sf) // in-place map, it overwrites the previous contents of vc

vc.i_map(sf) // in-place map, it overwrites the previous contents of vc

plot(v, vc, Color.GREEN) // plot also the overwritten v

We can easily compute the *sum, product* and *mean* of the elements of a vector:

v = V(7.8, 9.8, 10.9, -4.3)

sv = v.sum()

pv = v.prod()

mv = v.mean()

Also, we can compute some vector norms:

```
x= 9.3

v = V(x, sin(x), 7.8, 9.8, 10.9, -9.8, -3.4*exp(x))

norm1v = v.norm1()

norm2v = v.norm2()

norm2robust = v.norm2_robust()

norm2inf = v.normInf()
```

We can perform pointwise addition/subtraction/multiplication/division easily as:

```
v = V(7.8, 9.8, 10.9)

vpv = v+v

vmv = v-v

vmulv=  v*v

vdv = v/v
```

We negate the vector easily:

```
v2 = -v
```

We can raise the elements of a vector to the powers described by another vector pointwise as:

```
v = V(7.8, 9.8, 10.9)

vp = V(1, 2, 2)

vpow = v ** vp   // vector element-wise power operator
```

The *vones(), vzeros, vfill()* are also useful (and simple) routines:

```
v1 = vones(9)

vz  = vzeros(7)

vf = vfill(20, 7.8)
```

The *vlinspace(startv, endv)* routine creates a vector of 100 equally spaced points between startv and endv:

```
vv = vlinspace(0, 20)
```

We can explicitly specify the number of points with *vlinspace(startv. endv, npoints):*

vv = vlinspace(0, 20, 500)

Similarly, the *vlogspace(startv, endv)* routine creates a vector of 100 logarithmicallly spaced points between startv and endv (default logarithm base 10 is used):

vv = vlogspace(0, 20)

We can explicitly specify the number of points with *vlogspace(startv. endv, npoints):*

vv = vlogspace(0, 20, 500)

Also, the *vinc()* is useful since it samples a range with a fixed sampling interval, e.g.:

vv = vinc(0, 0.01, 10)

Tutorial on Matrix class

The Matrix class is very important since:

●It provides ***a lot of important static operations*** that cover important computational tasks, e.g. *eig(), svd(), solve(), etc.* Those operations are specified both for *double [][]* arrays and for *Matrix* objects, e.g. *svd(A)* is defined for *A* both a double [][] array and *A* being a Matrix object. Therefore by making a *static import* of the *Matrix* class we have conveniently available a lot of useful static methods, to work with.

●The *Matrix* objects are equipped with a lot of computational functionality. The usual operators are overloaded for Matrix objects, e.g. we can easily multiply Matrix *A* with *B* as *A*B.* However, in GroovyLab, using the *metaprogramming* facilities of Groovy, we overload operators and for the standard *double[][]* Java arrays, in order to work with them more elegantly.

This tutorial describes by means of examples the Matrix class of GroovyLab. The Matrix class implements zero-indexed two-dimensional dense matrices in GroovySci.

To recapitulate things better, the **Matrix** class is fundamental since:

a. it provides a lot of **mathematical operations** implemented using different Java libraries, for example for linear system solvers we can use either the JLAPACK based solvers, the EJML, the Apache Common Maths, the Numerical Recipes, the NUMAL or the JAMA one.

Some libraries as the *Apache Common Maths,* the *Numerical Recipes,* the *JAMA* library and the *NUMAL* library use a **2-D double array** matrix representation as the GroovyLab *Matrix* class also does. Therefore their routines are readily accessible without any conversion. Some other libraries use different representations, for example JLAPACK uses **1-D double array**, in which the matrix storage layout is in **column based order** (i.e. Fortran like). In these cases, *O(N)* conversion routines (*N* the number of matrix elements) are required before using algorithms of these libraries, therefore only mathematical routines of much higher than linear complexity (e.g. matrix inversion, Singular Valued Decomposition, eigenvalue computations) can perhaps benefit from such libraries.

b. it provides a lot of useful **static methods**, usually overloaded to work on many different types, e.g.:

Matrix sin(Matrix a), double [][] sin(double [][] a), double [] sin(double [] a) etc.

GroovyLab imports by default **all the static methods** of the Matrix class before any code is executed with the GroovyShell. Therefore we can write *sin(x),* where *x* can take many possible types, e.g. *Matrix, double [][], double [], double.*

Matrix Construction

We can construct a matrix **from its elements** conveniently using any expressions with the *Mat(int nrows, int ncols, double ...values)* method. For example:

```
x = 2.3; y = 8.9
mxy = Mat(2,3,   // a matrix of 2 rows and 3 columns
   x, -x*sin(x), x+tan(x),
   0.34*x, log(x+sin(y)), x*y*sin(x*y))
```

Another way to construct a matrix is usimg the constructor:

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

For example:

m=new Matrix(2, 3)

We can also construct a Matrix **from a double [][] array of values**, with the constructor *Matrix(double [][] da).* For example:

dd = new double[2][4]
dd[1][1]=11
mdd = new Matrix(dd)

We can now index the 11 as:

eleven = mdd[1,1]

or with direct access to the Java array as:

elevenJava = mdd.d[1][1]

We can also construct a matrix, specifying at the same time an initial value for all of its elements. This is accomplished with the constructor, *Matrix(int N, int M, double value),* e.g.

m = new Matrix(6, 8, 6.8)

Construction with specified values

We can construct a matrix using specified values in three ways.
1. Constructing a double [][] representation then convert to Matrix, e.g.
x = 7.7 // a value
*md = [[4.5*x, 5.6, -4-x], [4.5*cos(x), 3, -3.4], [14.5, 2.2+exp(x), -2.4], [40.5, 2.2, -3.4]] as double[][]*
m = new Matrix(md)

```
x1 = m[0, 1] // first row, second column
a = rand(30, 40)  // a 30 rows by 40 cols random matrix
x2 = a.grc(1, 2, 4, 2, 2, 10) // like MATLAB's  a(1:2:4, 2:2:10)
x3 = a.gr(1, 4)  // like MATLAB's a(1:4, :)
x4 = a.gc(1, 5)  // column select, all rows, as MATLAB's a(:, 1:5)
```

2.Using the static method *Mat()* of the *Matrix* class that returns a constructed Matrix object from variable argument double parameters. The first two parameters specify the number of rows and columns. For example:

x = 5.5; y = 73.3 // some values

A = Mat(3, 3, // number of rows, and columns

*3.4*x, x, -0.4, // first row*

*-cos(x+y), x- sin(x*y), x, // second row*

*x-0.3, x/x, y/(2*y) // third column*

)

3. The *M()* static method constructs a Matrix from a String specification, in which the matrix elements are either space or comma separated. The Matrix lines are separated with semicolons. For example:

a1 = M("4 5.9; 5.6 9.8")

a2 = M("47, -5.9; 0.56, 9.8")

However, a disadvantage of the *M()* method is that it operates only with numeric literals, i.e. we cannot have variables and expressions.

Matrix Indexing/Setting operations

The GroovyLab *Matrix* class stores data using a *double [][]* array. Generally, this representation is not the optimal, but it is simple and generally efficient. As a useful note, Java arrays storage is row based, and therefore is more efficient to work with work with fixed rows, i. e. the inner faster changed index should operate on columns.

We can perform indexing in three ways:

a. Exploiting the overloading of the array indexing operator [] by the methods *getAt(), putAt()* of Groovy, as for example:

A = rand(2,3) // a 2 X 3 Matrix

a12 = A[1,2] // get the element

A[1][2] = 2.3 // put the value

b. By directly accessing the Java double[][] internal representation, as for example:

A = rand(2,3) // a 2 X 3 Matrix

a12 = A.d[1][2] // get the element

A.d[1][2] = 2.3 // put the value

The second way, although not as elegant, is faster.

c. By using the ***gr(), gc(), grc()*** methods. We explain these methods:

- The group of overloaded methods named ***gr()*** (named by the initials **g**et **r**ow) selects rows of the matrix, i.e. for a specified group of rows, select all the columns.
- The group of overloaded methods named ***gc()*** (named by the initials **g**et **c**olumn) selects columns of the matrix, i.e. for a specified group of columns, select all the rows.
- The group of overloaded methods named ***grc()*** (named by the initials **g**et **r**ow-**c**olumn) selects rows and columns of the matrix, i.e. for a specified group of rows/columns, select all that matrix part.

Below we describe these methods for *matrix subrange chores* in detail.

Matrix subrange operations

These operations aim to provide convenient getting/setting of matrix ranges. They are implemented by the methods ***gr(), gc(), grc()*** for getting and ***sr(), sc(), src()*** for setting values. The design is kept simple and symmetrical. It is also efficient, since it is coded in Java.

It is better to understand these operations by means of examples.

Let construct an example matrix:

```
A = Mat( 6, 4,
      1.2, -0.2, 5.6, -0.03,
      4.5, 0.03, 2.3, 0.2,
     0.12, -2.2, 15.6, -11.3,
      5.34, 5.03, 1.3, 0.3,
      0.022, -120.2, 6.778, -10.03,
```

145, 10.03, 13, -0.2)

We proceed by describing the ***get style*** routines, and then the ***set style***ones.

Get Routines

Single row/column select

We can get a column of the matrix as:

Ac = A.gc(2) // get column 2

Similarly, for getting a row:

Ar = A.gr(1) // get row 1

or as

Ar = A[1] // get row 1

Select a continuous range of rows/columns

We can get a column range of the matrix as:

Acr = A.gc(1, 2) // get columns 1 to 2 all rows

Similarly, for getting a row range:

Arr = A.gr(1, 2) // get rows 1 to 2 all columns

or

Arr = A[1..2] // get rows 1 to 2 all columns

Additionally, we can specify an increment:

Acr = A.gc(0, 2, 3) // as MATLAB's A(:, 0:2:3)

or

Acr = A[-1..-1, (0..3).by(2)] // negative index specifies all

Similarly we can select rows specifying an increment:

Arr = A.gr(0, 2, 4) // as MATLAB's A(0:2:4, :)

or

Arr = A[(0..4).by(2)]

We can extract rectangular parts of the matrix as:

Arc = A.grc(0, 2, 4, 1, 1, 3) // as MATLAB's A(0:2:4, 1:1:3)

or

Arc = A[(0..4).by(2), (1..3).by(1)]

Specifying particular rows to extract

We can specify particular rows to extract as:

rowIndices = [3, 1] as int [] // specify the required rows in an int[] array

erows = A.gr(rowIndices)   // get the rows

Specifying particular columns to extract

Similarly, we can specify particular columns to extract as:

colIndices = [1, 3, 2] as int [] // specify the required columns in an int[] array

ecols = A.gc(colIndices)   // get the columns

Extracting submatrices using Groovy ranges

Groovy ranges allow a nice syntax. We can use them as illustrated:

Ar = A.gr(1..2)   // get rows 1 to 2

or

Ar = A[1..2] // get rows 1 to 2

Ac = A.gc(2..3) // get columns 2 to 3

or

A[-1..1,2..3] // get columns 2 to 3

Arc = A.grc(1..2, 2..3) // get matrix subrange of rows 1 to 2 and columns 2 to 3

or

Arc = A[1..2, 2..3]   // get matrix subrange of rows 1 to 2 and columns 2 to 3

Set Routines

Copy a matrix within another matrix

x = rand(12, 15) // a random matrix

y = ones(2, 3)   // a matrix of 1s

x.s(2, 2, y) // copy the matrix of 1s at the 2,2 position within the random matrix

Set a row range to a value

a = rand(5,9) // a random matrix

a.sr(1..2, 8)   // sets rows 1 and 2 to 8

Set a row submatrix to a value

a = rand(5,9) // a random matrix

a.sr(1, 2, 8)   // sets rows 1 and 2 to 8

Set a column range to a value

a = rand(5, 9) // a random matrix

a.sc(2..3, 77.7) //sets cols 2 to 3 to 77.7

Set a column submatrix to a value

a = rand(5, 9) // a random matrix

a.sc(2, 3, 77.7) // sets cols 2 to 3 to 77.7

Set a row-column range to a value

a = rand(10, 15) // a random matrix

a.src(2..5, 1..3, 11.2) // sets rows 2 to 5 and columns 1 to 3 to 11.2

Set a row-column matrix part to a value

a = rand(10, 15) // a random matrix

a.src(2, 5, 1, 3, 11.2) // sets rows 2 to 5 and columns 1 to 3 to 11.2

Set operations using ranges

We can use ranges to perform set operations more conveniently. We present some illustrative examples:

// Example 1

a = rand(10, 15) // a random matrix

a[1..3] = 1.1 // set rows 1 to 3 to 1.1

// Example 2

a = rand(10, 15) // a random matrix

a[(1..5).by(2)] = 1.1 // as a(1:2:5, :) = 1.1

// Example 3

a = rand(10, 15) // a random matrix

a[-1..-1, 1..3] = 19.1 // as a(:. 1:3) = 19.1

// Example 4

a = rand(10, 15) // a random matrix

a[-1..-1, (1..9).by(2)] = 59.1 // as a(:. 1:2:9) = 59.1

// Example 5

a = rand(10, 15) // a random matrix

a[1..3, 2..4] = 1.1 // as a(1:3. 2:4) = 1.1

// Example 6

a = rand(10, 15) // a random matrix

a[(1..10).by(3), (1..9).by(2)] = -59.1 // as a(1:3:10. 1:2:9) = -59.1

Operators

The following operators are available: Matrix + Matrix, Matrix + Number, Matrix - Matrix, Matrix - Number, Matrix * Matrix, Matrix * Number, Matrix / Matrix, Matrix / Number, Number + Matrix, Number * Matrix

```
a = rand(5, 8)
aa = a+a
a5 = a+5
ap5 = 5+a
a6 = a*6.8
a6m = 6.8*a
a85 = rand(8, 5)
ama = a*a85
```

More convenient access/assignment operations

We can use and more elegant syntactially constructs to access/update submatrices. For example:

```
x = rand(80, 100)
x[2..3, 0..1] = 6.5353  // a rectangular subrange
x
x[4..5] = -0.45454  // assign rows 4 to 5 all columns

// assignment using ranges
x = rand(90, 90)
x[(0..20).by(2), 0..1] = 99

// assignment using ranges
x = rand(90,90)
x[0..1, (0..30).by(3)] = 33.3

// assignment using ranges
x = rand(90,90)
x[(2..14).by(5), (0..30).by(3)] = -77.3

row=2; col = 3
y = x[row..row+2, col..col+5]  // get a matrix range
```

yy = x[(row..row+50).by(2), (col..col+30).by(3)] // like MATLAB's x(row:2:row+50, col:3:col+30)

Static operations

Some **static operations** that are provided with the **Matrix** class are:

```
oo = ones(4, 10)  // a matrix with all ones
oo4 = sum(oo)  // perform sum of the columns
sum(sum(oo))  // total sum of the matrix elements
oo2 = fill(4, 10, 2.0) // a matrix filled with 2 values
prod(oo2) // perform product of the columns
prod(prod(oo2)) // product of all the matrix elements
csoo = cumsum(oo) // perform a cummulative sum across columns
cpoo2 = cumprod(oo2)  // perform a cummulative product across columns
aa = rand(5,5)
aai = inv(aa)  // compute the matrix inverse
A = rand(5,5)
b = rand(5,1)
X = solve(A, b) //returns X Matrix verifying A*X = b.
rank(A) // the rank of A
trace(A) // the trace of A
det(A)  // the determinant of A
cond(A) // the condition number of A
norm1(A) // norm 1 of A
norm2(A) // norm 2 of A
normF(A) // Frobenius norm of A
normInf(A) // norm inf of A
dot(A, A) // the dot product of A by itself
```

Numerical Analysis with GroovyLab

This section demonstrates how we can perform many common numerical analysis tasks with GroovyLab by means of simple examples.

Computing the determinant of a matrix

```
a = Mat(2, 2, 1, 2, 3, 4)   // as Matrix
da = det(a)

ad = [[1, 2], [ 3, 4]] as double [][]   // as double [][] array
dda = det(a)
```

Computing the inverse of a matrix

```
a = Mat(4,4,
     1, -1, 1, 2,
      1, 0, 1, 3,
      0, 0, 2, 4,
      1, 1, 0, -1)
ai = inv(a)
a*ai-diag(4)/4   // this matrix should display all zeros
Mxelem = max(max(a*ai-diag(4)/4))  // should be a very small value
```

Computing norms

```
a = Mat(4, 4,
  1, -1, 1, 2,
  1, 0, 1, 3,
  0, 0, 2, 4,
  1, 1, 0, -1)
norm1(a)  //  the largest column sum
norm2(a)  // maximum singular value
```

```
normF(a)  // the Frobenious norm, sqrt(sum(diag(x'*x)))
normInf(a)   //  the infinity norm, the largest row sum
rank(a)  // the rank of a
svd_rank(a)  // the rank of a computed with SVD
trace(a)  // the trace of a
```

GroovySci Examples

In this section we provide small examples of using GroovySci, in scientific applications. The code can be executed either:

- by pasting it at the GroovyLab's command console
- by pasting the code within the GroovyLab's programmer's editor, select the code and pressing F6 or F8

Henon Chaotic Map

This small GroovySci script computes some iterations of the Henon chaotic map and plots them. Note here that by seting ***Transforms BigDecimals to Doubles*** to true (from the "***Configuration***" menu), this computation becomes much faster.

```
x = 0.0; y = 0.0
niters = 10000
alpha = 1.4
beta = 0.3
tic()
xy=new double[2][niters]
for (k=1; k<niters; k++)  {
  xp = x;yp=y
  x = 1+yp-alpha *xp*xp
  y = beta*xp
```

```
  xy[0][k] = x
  xy[1][k] = y
}
scatterPlotsOn()    // plot points without connecting with lines
plot(xy)
```

Baker map

```
x = 0.1;  y = 0.22;
niters = getInt("How many iterations of the Baker map");
xall = new double[niters]
yall = new double[niters]
y13 = 1.0/3.0
y23 = 2.0/3.0
tic()
for (k in 1..niters-1)  {
  xp = x; yp=y
 if (y<=0.5) {
   y = 2*yp
   x = y13*xp
}
else {
  x = y13*xp+y23
  y = 2*yp - 1
  }
  xall[k] = x
  yall[k] = y
}
tm = toc()
scatterPlotsOn()
figure(1)
plot(xall, yall, "time = "+tm)
```

Ikeda Chaotic Map

```
// the Ikeda map
 R = 1;  C1 = 0.4; C2 = 0.9;  C3 = 6
niters = getInt("How many iterations of the Ikeda map")
x = new double[niters]
y = new double[niters]
x[0]=0.12; y[0]=0.2
tic()
k=1
km=0
 tau=0.0;  sintau=0.0;  costau=0.0
while  (k< niters)  {
  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++
}
tm = toc()
scatterPlotsOn()
figure(1)
plot(x, y, "time = "+tm)
```

Demonstrating surface plots

```
// demonstrate a surface plot
 // below is the script code. It is wrapped to a class of name surfPlot by Groovy
 X = inc(-2, 0.2, 2)
 Y = inc(-2, 0.2, 2)

 x = X.getArray()[0]
 y = Y.getArray()[0]
```

```
z1 = Functions.f1(x, y)
z2 = Functions.f2(x, y)
figure3d(1);  surf(x, y, z1, "Surface Plot")
figure3d(2);  surf(x, y, z2, "Surface Plot 2")
figure3d(3);  surf(x, y, z1, "Surface Plot"); surf(x, y, z2, "Surface Plot 2")
class Functions {
    // function definition: z=cos(PI*x)*sin(PI*y)
    public static double f1(double x, double y) {
        double z = Math.cos(x * Math.PI) * Math.sin(y *Math.PI)
        return z
    }

    // grid version of the function
    public static double[][] f1(double[] x, double[] y) {
        double[][] z = new double[y.length][x.length]
        for (int i = 0; i < x.length; i++)
            for (int j = 0; j < y.length; j++)
                z[j][i] = f1(x[i], y[j])
        return z
    }

    // another function definition: z=sin(PI*x)*cos(PI*y)
    public static double f2(double x, double y) {
        double z = Math.sin(x * Math.PI) * Math.cos(y * Math.PI)
        return z
    }

    // grid version of the function
    public static double[][] f2(double[] x, double[] y) {
        double[][] z = new double[y.length][x.length]
        for (int i = 0; i < x.length; i++)
            for (int j = 0; j < y.length; j++)
                z[j][i] = f2(x[i], y[j])
        return z
```

```
    }
 }
```

A cloud plot demo

```
figure(1)
N = 1000
cloud = new double[N][2]
for (int i = 0; i < cloud.length; i++) {
        cloud[i][0] = N * Math.exp(Math.random() + Math.random())
        cloud[i][1] = N * Math.exp(Math.random() + Math.random())
}
slicesX = 2
slicesY = 2
plotname = "Demo2d cloud_plot "
plot2d_cloud(cloud, slicesX, slicesY, plotname)
```

Perform some plots that test the VISAD interface, requires installation of Java 3D

```
//demonstrate MATLAB-like plotting using VISAD
  // create signals
    t=inc(0, 0.01, 10); x = sin(3.4*t)
    y = sin(0.48*t)
    z = sin(5.7*x+0.2*y)
    vfigure(1);
    vsubplot(2,1,1);
    vplot(x, y, z)  // 3-D plot
    vsubplot(2,1,2);
    vplot(x, y, sin(9.8*z))
    vfigure(2);
    vsubplot(2, 2, 1);   vplot(x, 8);  // plot with 8 point line
    vsubplot(2, 2, 2);   vplot(x)
```

```
vsubplot(2, 2, 3);   vplot(sin(0.89*x))
zz = z+cos(0.8*z)
vsubplot(2, 2, 4);  vplot(z);
vaddplot(zz)   // add the plot without erasing previous, i.e. in "hold on" state
zzz = zz+ sin(3.4*zz)
vaddplot(zzz, "zzx", "zzy", 5)

vfigure(3);
vsubplot(2, 2, 1);   vplotPoint(x);  // plot with 8 point line
vsubplot(2, 2, 2);   vplot(x)
vsubplot(2, 2, 3);   vplotXYPoints(x, y, "x", "y", 8)
vsubplot(2, 2, 4);  vplot(z)
vaddplot(zz)   // add the plot without erasing previous, i.e. in "hold on" state
vaddplot(zzz, "zzx", "zzy", 5)
```

A Continuous Wavelet Transform example

```
clear("all")
fs = 600; // 2050
dt = 1/fs
t =  inc(1, dt, 4)
PI2 = 2*PI
y = sin(PI2*10*t)+4*cos(PI2*4*t)
y = y+5*rand(y.size()[0], y.size()[1])
y
N = y.size()[1]
fstart = 1;  // frequency to start
fmax = (double)fs/2
maxNf = 20
linlog = "log"
stepfac=16
df0=3
ycwt = new wavelets.CWT(y,  fs, fmax, maxNf, linlog, stepfac, df0)
ed = ycwt.ed()  // energy density coefficients as a double[][] vector
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(subsampledEdm, "scalogram")
```

Plotting using a MATLAB-like interface to the JFreeChart library

```
jfigure(1)
t = inc(0, 0.01, 10);   x = sin(0.23*t)
lineSpecs = "."
jplot(t,x, lineSpecs)
jtitle("drawing multiple line styles")
jhold(true)  // hold axis
lineSpecs = ":r+"
jplot(t, 0.1*cos(9.8*x), lineSpecs)
// redefine the color of line 2
jlineColor(2, Color.BLUE)
jfigure(2)
jsubplot(222)
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
 nf = jfigure()
```

```
jsubplot(3,2,1)
t2 = inc(0, 0.01, 10);  x2 = sin(3.23*t2)+2*cos(0.23*t2)
jplot(t2,x2, ".-")
jsubplot(3,2,3)
x3 = cos(2.3*t2)+9*sin(4.5*t2)
jplot(t2, x3)
jlineColor(1, Color.RED)
jsubplot(3,2,5)
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
jhold(true);
jsubplot(2, 2, 1)
vr = rand(1, 2000)
jplot(vr)
jtitle("A Random Vector")
td = t.getv()
jsubplot(224)
jplot(td, sin(1.34*td))
jplot( td, sin(3.6*td))
jtitle("Multiple Plots")
// demonstrate PieDataChart
c = new String[3];  c[0] = "Class1"; c[1] =  "Class2"; c[2] = "Class3"
v = new double[3]; v[0]=5.7; v[1] = 9.8;  v[2] = 3.9
pieChartName = "Test Pie Chart"
myPie = jplot(pieChartName, c, v)
```

Plotting Demonstrations

```
    t = inc(0, 0.01, 20)
    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")
    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 = inc(0, 0.01, 20)
     y = sin(0.2*t)+5*cos(0.23*t)
     z = sin(1.2*t)+0.5*cos(0.23*t)
    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")
    zDot = dot(z, z)
    fig = figure3d(3); plot(t,y, zDot, Color.RED)
// 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))")
    zDot2 = dot(zDot, zDot)
    fig = figure3d(4)
    subplot3d(2,1,1)
    plot(t,y, zDot, 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, zDot, 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("zDot")
```

And other plotting Demonstrations

```
// to run this demo copy and paste the code
t=inc(-10, 0.01, 20)   // create a time axis
freq = 2
x=sin(freq*t)  // a simple sinusoid
y = 70* cos(0.4*t)+12*sin(0.7*t)
z = dot(y, 0.01*t)   // dot product
h = dot(z, sin(0.3*x))   //  if the operator * cannot perform matrix multiplication it tries to perform dot product
figure();  plot(t,x, "t -  sin(t)")
figNew = figure();  subplot(2,1,1);  plot(t, z, "t - t*x : A sine supeimposed on a linear inreasing curve");
subplot(2,1,2);  plot(z, h, "y .* 0.01*t, z .* sin(0.3*x)")
figure(); hold("off"); subplot(2,1,1);  hold("on");  plot(x,y, Color.RED); plot(x,z, Color.BLUE);
subplot(2,1,2);   hold("off"); plot(x,y, Color.RED); plot(x,z, Color.BLUE);
 title("Demo for hold(\"on\"): up subplot,  hold(\"off\"), bottom subplot");
sample = rand(3, 1000);
slices_x = 6; slices_y = 6; slices_z = 6; name="cloud";
 plot3d_cloud( sample, slices_x, slices_y, slices_z,  name);
```

Various sine functions on the same plot

```
closeAll()
dt = 0.01;  // sampling frequency
xl = -10;  xu = 20;  // low and up limits
```

```
t=inc(xl, dt, xu);   // time axis
f11 = 0.23;  f12 = 3.7;  // two frquencies
f21 = 0.25;  f22 = 3.9;  // slightly different frquencies
x1 = sin(f11*t) + cos(f12*t);
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");
x12 = dot(x1, x2);
plot(t, x12, Color.BLUE, "x1 .* x2");
```

Using the Groovy's SwingBuilder to specify signal parameters and process a signal

```
// demonstrates using swing builder for getting signal parameters
// and performing FFT on that signal
swing = new SwingBuilder()
 frame = swing.frame(title:'Parameter Screen for signal') {
panel {
    label('Sine Frequency')
    textField(id:'frequency',  columns:10)
    label('Noisy energy')
    textField(id:'noiseCoef', columns:10)
    button(text: 'Get Parameters', actionPerformed: {
    f1 =Double.valueOf(swing.frequency.text)
    noiseCoeff = Double.valueOf(swing.noiseCoef.text)
    closeAll()
    figure(1)
    t = inc(0, 0.02, 90)
    n = 2**12
    t = t[0..0, 0..n-1]
    x = sin(f1*t)+noiseCoeff*rand(1,n)
    y = x+cos(x)  // processed signal
    figure(1)
    title("signal and its processed signal")
    subplot(2,1,1); plot(x, Color.RED, "Signal");
    subplot(2,1,2); plot(y, Color.BLUE, "Processed Signal");
```

```
    })
  }
}
frame.pack()
frame.show()
```

Some simple signals with induced noise

```
t = inc(-5, 0.01, 10)
f1 = 0.34
f2 = 0.56
x = sin(f1*t)+5*cos(f2*t)
plot(t,x, "t-x plot")
y  = cos(x)
figure();   plot(x,y, "x-y plot")
sz=size(x)
xn = x+rand(sz[0], sz[1])
fign = figure()
plot(t, xn, "noisy signal")
```

Functional Plotting and plotting functions specified as expressions

Closures in Groovy is a powerful feature that allows to explore a functional programming style. We describe how closures are used to implement convenient functional style plotting in GroovyLab.

One dimensional function plotting

We can easily plot a **function** that is implemented with a **closure** using the ***fplot()***. This function has the prototype:

public static void fplot(Closure f1d, double low, double high, String title, java.awt.Color color, boolean linePlotsFlag, int nP)

The *f1d* parameter is the ***Closure*** that implements our 1-dimensional function. The *low* and *high* parameters define the limits over which the function will be plotted. The parameter *title* is the title

for the plot and *color* defines its color. Also, *linePlotsFlag* is a boolean parameter that defines if the points will be joined with a line. Finally, parameter *nP* is the number of points.

For example:

```
cube = { x -> x*x*x}   // a cube function
low = -5;  high= 5; color = Color.RED; linePlotsFlag = false; nP = 2000
fplot(cube, low, high, color, linePlotsFlag, nP)
```

We can easily plot a function specified with a String expression, using the similar ***splot()*** function. For example:

```
cubeSine = " x*x*x*sin(x)"   // a function
low = -5;  high= 5; color = Color.RED; linePlotsFlag = false; nP = 2000
figure(1); subplot(2,1,1);
splot(cubeSine, low, high, cubeSine, color, linePlotsFlag, nP)
squareCos = "x*x*cos(12.3*x)"
 low2 = -15;  high2= 15
subplot(2,1,2)
splot(squareCos, low2, high2, squareCos); // use defaults
```

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.

Note: The code for the *PlotController* object exists in GroovyLab, therefore we can execute directly the testing code that follows. It is presented for illustration purposes.

Design of a Plot Facilitator Object (this code exists in GroovyLab source)

```
  package groovySci.math.plot;
import java.awt.Color;
```

```
// a class to plot in an object-oriented way
public class PlotController  {


// these routines affect AbstractDrawer properties and can be used to change the defaults
// for external plotting routines that do not specify their own values
     // set drawing to continuous lines
   void  setContinuousLine()  { groovySci.math.plot.render.AbstractDrawer.line_type =
groovySci.math.plot.render.AbstractDrawer.CONTINOUS_LINE; }
     // set drawing to dotted lines
   void  setDottedLine()  { groovySci.math.plot.render.AbstractDrawer.line_type =
groovySci.math.plot.render.AbstractDrawer.DOTTED_LINE; }
     // set drawing to Round Dot
   void  setRoundDot()  { groovySci.math.plot.render.AbstractDrawer.dot_type =
groovySci.math.plot.render.AbstractDrawer.ROUND_DOT; }
     // set drawing to cross dot
   void  setCrossDot()  { groovySci.math.plot.render.AbstractDrawer.dot_type =
groovySci.math.plot.render.AbstractDrawer.CROSS_DOT; }
     // set font
   void  setFont(java.awt.Font font)  { groovySci.math.plot.PlotGlobals.defaultAbstractDrawerFont
= font; }
     // set Color
   void  setColor(java.awt.Color  color)
{ groovySci.math.plot.PlotGlobals.defaultAbstractDrawerColor = color; }
     // set line width
   void  setlineWidth(int lw)   { groovySci.math.plot.render.AbstractDrawer.line_width = lw; }

    // references to the data for plot
   double []  xData = new double [1];
   double []  yData = new  double [1];
   double []  zData = new double [1];

    // adjust the data references for plotting to the actual data
   void   setX( double [] x )  { xData = x; }  // sets the x-data for plotting
   void   setY( double [] y )  { yData = y; }  // sets the y-data for plotting
   void   setZ( double [] z )   { zData = z;  }  // sets the z-data for plotting

   void  setX( groovySci.math.array.Vec  x)  { xData = x.getv(); }  // sets the x-data using Vec
```

```
void  setY( groovySci.math.array.Vec  y)  { yData = y.getv(); }  // sets the y-data using Vec
void  setZ( groovySci.math.array.Vec  z)  { zData = z.getv(); }  // sets the z-data using Vec

void  setX( groovySci.math.array.Matrix  x)  { xData = x.getv(); }  // sets the x-data using Matrix
void  setY( groovySci.math.array.Matrix  y)  { yData = y.getv(); }  // sets the y-data using Matrix
void  setZ( groovySci.math.array.Matrix  z)  { zData = z.getv(); }  // sets the z-data using Matrix

String  xlabelStr = "X-axis";
void  setxlabel( String  xl )  {  xlabelStr = xl; }

String   ylabelStr = "Y-axis";
void  setylabel( String yl )  { ylabelStr = yl; }

String  zlabelStr = "Z-axis";
void  setzlabel( String zl )  {  zlabelStr = zl; }

String   plotTitle2D = "2-D Plot";
void   setplotTitle2D(String  plTitle)  { plotTitle2D = plTitle; }
String   plotTitle3D = "3-D Plot";
void   setplotTitle3D(String  plTitle)  { plotTitle3D = plTitle; }


// perform the plot using the object's properties
void mkplot()  {
    groovySci.math.plot.plot.plot(xData, yData,
groovySci.math.plot.PlotGlobals.defaultAbstractDrawerColor, plotTitle2D);
    groovySci.math.plot.plot.xlabel(xlabelStr);
    groovySci.math.plot.plot.ylabel(ylabelStr);


   }

// perform the plot using the object's properties
void  mkplot3D()  {


    groovySci.math.plot.plot.plot(xData, yData, zData,
groovySci.math.plot.PlotGlobals.defaultAbstractDrawerColor,  plotTitle3D);
```

```
    groovySci.math.plot.plot.xlabel(xlabelStr);
    groovySci.math.plot.plot.ylabel(ylabelStr);
    groovySci.math.plot.plot.zlabel(zlabelStr);
   }
}
```

Testing the Plot Controller Object

We can test the plot controller object as:

```
// Illustrates how to  construct a PlotController object to facilitate the plotting
 po = new PlotController()

// construct some signals
 x = inc(0, 1, 100)
 y1 = sin(0.45*x)
 y2 = sin(0.778*x)+0.2*cos(3.4*x)
 y3 = cos(y1+y2)

// construct the first plot object
closeAll()  // close any previous figure
po.setX(x)
po.setY(y1)
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.BLUE)
po.setlineWidth(5)   // set to thicker width
// redefine the new signals for plotting
hold(true)
po.setY(y1)   // we change only the Y signal
```

```
figure(); po.mkplot()

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()
```

The jzy3D for scientific plotting in GroovyLab

The jzy3D project system allows high quality plotting of data. In GroovyLab it is easy to use that system and to combine it with the other libraries. Here we present examples of its use both with its native interface and with a MATLAB-like one that we develop.

3-D Plotting of a function

```
import org.jzy3d.chart.Chart
import org.jzy3d.colors.Color
import org.jzy3d.colors.ColorMapper
import org.jzy3d.colors.colormaps.ColorMapRainbow
import org.jzy3d.demos.AbstractDemo
import org.jzy3d.demos.Launcher
import org.jzy3d.maths.Range
import org.jzy3d.plot3d.builder.Builder
import org.jzy3d.plot3d.builder.Mapper
import org.jzy3d.plot3d.builder.concrete.OrthonormalGrid
import org.jzy3d.plot3d.primitives.Shape
import org.jzy3d.plot3d.rendering.legends.colorbars.ColorbarLegend


// define a function to plot
 mapper = new Mapper(){
          double  f(double x,  double  y )   {
    return   10*Math.sin(x/10)*Math.cos(y/20)*x
```

```
        }
    }

// define range and precision fort the function to plot
range = new Range(-150, 150)

steps = 50

// create the object to represent the function over the given range
surface = (Shape) Builder.buildOrthonormal(new OrthonormalGrid(range, steps, range, steps),
mapper)

surface.setColorMapper(new ColorMapper(new ColorMapRainbow(),
surface.getBounds().getZmin(), surface.getBounds().getZmax(),
new Color(1,1,1, 0.5f)))

surface.setFaceDisplayed(true)

surface.setWireframeDisplayed(true)
surface.setWireframeColor(Color.BLACK)

// create a chart and add surface
chart = new Chart(org.jzy3d.plot3d.rendering.canvas.Quality.Advanced)

chart.getScene().getGraph().add(surface)

// setup a colorbar
cbar = new ColorbarLegend(surface, chart.getView().getAxe().getLayout())
surface.setLegend(cbar)

org.jzy3d.ui.ChartLauncher.openChart(chart)
```

A Matlab like interface for plotting with jzy3D

A Matlab-like interface for plotting with jzy3D is in development, implemented with the jzy3dPlot.jzy3dPlot class. This is in a very early stage, however here is a simple plotting example

using this class.

```
N = 200
x =  linspace(0, 10, N)
F1 = 3.3; F2 = 4.5; A1= 2.3; A2 = 0.44;
y= A1*sin(F1*x)+A2*cos(A2*x)
z =  cos(0.678*y)

plot3d(x.v, y.v, z.v)
```

And another example for scatterplots.

```
N = 200
x =  linspace(0, 10, N)
F1 = 3.3;  F2 = 4.5;  A1= 2.3;  A2 = 0.44;
y = A1*sin(F1*x)+A2*cos(A2*x)
ZF1 = 3.3;  ZF2 = 4.5; ZA1= 2.3; ZA2 = 0.44;
z = ZA1*sin(ZF1*x)+ZA2*cos(ZA2*x)
scatter3d(x.getv(), y.getv(), z.getv(), new java.awt.Rectangle(100, 100, 500, 500), "Scatter Plot", 1)
```

Executing a Java example (with F9 keystroke)

Java examples of jzy3D can be executed easily from the editor with the F9 keystroke. Java classes should not be placed in a package. We present some examples.

Animated Surface (Java code - F9 keystroke to execute)

```
import org.jzy3d.demos.animation.ParametrizedMapper;

import java.awt.Graphics;
```

```
import java.awt.Graphics2D;
import java.util.List;

import org.jzy3d.chart.Chart;
import org.jzy3d.colors.Color;
import org.jzy3d.colors.ColorMapper;
import org.jzy3d.colors.colormaps.ColorMapRainbow;
import org.jzy3d.demos.AbstractDemo;
import org.jzy3d.demos.IRunnableDemo;
import org.jzy3d.demos.Launcher;
import org.jzy3d.maths.Coord3d;
import org.jzy3d.maths.Range;
import org.jzy3d.maths.TicToc;
import org.jzy3d.maths.Utils;
import org.jzy3d.plot3d.builder.Builder;
import org.jzy3d.plot3d.builder.Mapper;
import org.jzy3d.plot3d.builder.concrete.OrthonormalGrid;
import org.jzy3d.plot3d.primitives.AbstractDrawable;
import org.jzy3d.plot3d.primitives.Point;
import org.jzy3d.plot3d.primitives.Polygon;
import org.jzy3d.plot3d.primitives.Shape;
import org.jzy3d.plot3d.rendering.legends.colorbars.ColorbarLegend;
import org.jzy3d.plot3d.rendering.view.Renderer2d;


public class AnimatedSurfaceDemo extends AbstractDemo implements IRunnableDemo{

	public static void main(String[] args) throws Exception{
		IRunnableDemo demo = new AnimatedSurfaceDemo();
		Launcher.openDemo(demo);
		demo.start();
	}

	public AnimatedSurfaceDemo(){
		mapper = new ParametrizedMapper(0.9){
			public double f(double x, double y) {
```

```
                    return 10*Math.sin(x*p)*Math.cos(y*p)*x;
                }
            };
            Range range = new Range(-150,150);
            int steps   = 50;

            // Create the object to represent the function over the given range.
            surface = (Shape)Builder.buildOrthonormal(new OrthonormalGrid(range, steps,
range, steps), mapper);
            surface.setColorMapper(new ColorMapper(new ColorMapRainbow(),
surface.getBounds().getZmin(), surface.getBounds().getZmax(), new Color(1,1,1,.5f)));
            surface.setFaceDisplayed(true);
            surface.setWireframeDisplayed(true);
            surface.setWireframeColor(Color.BLACK);

            // Create a chart
            chart = new Chart("awt");
            surface.setLegend(new ColorbarLegend(surface,

        chart.getView().getAxe().getLayout().getZTickProvider(),

        chart.getView().getAxe().getLayout().getZTickRenderer()));
            chart.getScene().getGraph().add(surface);

            // display FPS
            fpsText = "";
            chart.addRenderer(new Renderer2d(){
                public void paint(Graphics g) {
                    Graphics2D g2d = (Graphics2D)g;
                    g2d.setColor(java.awt.Color.BLACK);
                    g2d.drawString(fpsText, 50, 50);
                }
            });
        }

        public Chart getChart(){
            return chart;
```

```
	}

	public void start(){
		fpsText = "";
		t = new Thread(){
			TicToc tt = new TicToc();
			@Override
			public void run() {
				while(true){
					try {
						sleep(25);
					} catch (InterruptedException e) {
						e.printStackTrace();
					}
					tt.tic();
					mapper.setParam( mapper.getParam() + 0.0001 );
					remap(surface, mapper);
					//chart.render();
					tt.toc();
					fpsText = Utils.num2str(1/tt.elapsedSecond(), 4) + "
FPS";
				}
			}
		};
		t.start();
	}

	public void stop(){
		if(t!=null)
			t.interrupt();
	}

	protected void remap(Shape shape, Mapper mapper){
		List<AbstractDrawable> polygons = shape.getDrawables();
		for(AbstractDrawable d: polygons){
			if(d instanceof Polygon){
```

```
                        Polygon p = (Polygon) d;
                        for(int i=0; i<p.size(); i++){
                            Point pt = p.get(i);
                            Coord3d c = pt.xyz;
                            c.z = (float) mapper.f(c.x, c.y);
                        }
                    }
                }
            }

            protected Chart chart;
            protected Shape surface;
            protected ParametrizedMapper mapper;
            protected String fpsText;

            protected Thread t;
}
```

Contour Demo (Java code - F9 keystroke to execute)

```
import org.jzy3d.chart.Chart;
import org.jzy3d.colors.ColorMapper;
import org.jzy3d.colors.colormaps.ColorMapRainbow;
import org.jzy3d.contour.MapperContourPictureGenerator;
import org.jzy3d.demos.AbstractDemo;
import org.jzy3d.demos.Launcher;
import org.jzy3d.maths.Coord3d;
import org.jzy3d.maths.Range;
import org.jzy3d.plot3d.builder.Mapper;
import org.jzy3d.plot3d.primitives.MultiColorScatter;
import org.jzy3d.plot3d.rendering.legends.colorbars.ColorbarLegend;
```

```
public class Contour3DDemo extends AbstractDemo{

    public static void main(String[] args) throws Exception{

        Launcher.openDemo(new Contour3DDemo());

    }



    public Contour3DDemo(){

        // Define a function to plot

        Mapper mapper = new Mapper(){

            public double f(double x, double y) {

                return 10*Math.sin(x/10)*Math.cos(y/20)*x;

            }

        };



        // Define range and precision for the function to plot

        Range xrange = new Range(-100,100);   //To get some more detail.

        Range yrange = new Range(-100,100);   //To get some more detail.
```

```
// Compute an image of the contour

MapperContourPictureGenerator contour = new MapperContourPictureGenerator(mapper, xrange, yrange);

int nPoints= 1000;

double[][] contours= contour.getContourMatrix(nPoints, nPoints, 40);



// Create the dot cloud scene and fill with data

int size = nPoints*nPoints;

Coord3d[] points = new Coord3d[size];



for (int x = 0; x < nPoints; x++)

    for (int y = 0; y < nPoints; y++){

        if (contours[x][y]>-Double.MAX_VALUE){ // Non contours points are -Double.MAX_VALUE and are not painted

            points[x*nPoints+y] = new Coord3d((float)x,(float)y, (float)contours[x][y]);

        }

        else

            points[x*nPoints+y] = new Coord3d((float)x,(float)y, (float)0.0);
```

```
//                                      points[x*400+y] = new Coord3d((float)x,(float)y,
(float)mapper.f(x, y));

                              }

                    chart = new Chart();

                    MultiColorScatter scatter = new MultiColorScatter( points, new
ColorMapper( new ColorMapRainbow(), -600.0f, 600.0f ) );

                    chart.getScene().add(scatter);

                    scatter.setLegend( new ColorbarLegend(scatter,

                                        chart.getView().getAxe().getLayout().getZTickProvider(),

                                        chart.getView().getAxe().getLayout().getZTickRenderer()) );

                    scatter.setLegendDisplayed(true);

          }

          public Chart getChart(){

                    return chart;

          }

          protected Chart chart;

}
```

Contour Plots Demo   (Java code - F9 keystroke to execute)

```
import org.jzy3d.chart.Chart;
import org.jzy3d.colors.Color;
import org.jzy3d.colors.ColorMapper;
import org.jzy3d.colors.colormaps.ColorMapRainbow;
import org.jzy3d.contour.DefaultContourColoringPolicy;
import org.jzy3d.contour.MapperContourPictureGenerator;
import org.jzy3d.demos.AbstractDemo;
import org.jzy3d.demos.Launcher;
import org.jzy3d.factories.JzyFactories;
import org.jzy3d.maths.Range;
import org.jzy3d.plot3d.builder.Builder;
import org.jzy3d.plot3d.builder.Mapper;
import org.jzy3d.plot3d.builder.concrete.OrthonormalGrid;
import org.jzy3d.plot3d.primitives.Shape;
import org.jzy3d.plot3d.primitives.axes.AxeFactory;
import org.jzy3d.plot3d.primitives.axes.ContourAxeBox;
import org.jzy3d.plot3d.primitives.axes.IAxe;
import org.jzy3d.plot3d.rendering.canvas.Quality;
import org.jzy3d.plot3d.rendering.legends.colorbars.ColorbarLegend;


public class ContourPlotsDemo extends AbstractDemo{
	public static void main(String[] args) throws Exception{
		Launcher.openDemo(new ContourPlotsDemo());
	}

	public ContourPlotsDemo(){
		// Define a function to plot
```

```
Mapper mapper = new Mapper(){
        public double f(double x, double y) {
                return 10*Math.sin(x/10)*Math.cos(y/20)*x;
        }
};

// Define range and precision for the function to plot
Range xrange = new Range(50,100);
Range yrange = new Range(50,100);
int steps   = 50;
final Shape surface = (Shape)Builder.buildOrthonormal(new
OrthonormalGrid(xrange, steps, yrange, steps), mapper);
ColorMapper myColorMapper=new ColorMapper(new ColorMapRainbow(),
surface.getBounds().getZmin(), surface.getBounds().getZmax(), new Color(1,1,1,.5f));
surface.setColorMapper(myColorMapper);
surface.setFaceDisplayed(true);
surface.setWireframeDisplayed(true);
surface.setWireframeColor(Color.BLACK);

// Create a chart with contour axe box, and attach the contour picture
JzyFactories.axe = new AxeFactory(){
        @Override
        public IAxe getInstance() {
                return new ContourAxeBox(box);
        }
};
chart = new Chart(); //TODO: Quality.Advanced contour buggy with axe box
ContourAxeBox cab = (ContourAxeBox)chart.getView().getAxe();
MapperContourPictureGenerator contour = new
MapperContourPictureGenerator(mapper, xrange, yrange);
cab.setContourImg( contour.getContourImage(new
DefaultContourColoringPolicy(myColorMapper), 400, 400, 10), xrange, yrange);

// Add the surface and its colorbar
chart.addDrawable(surface);
surface.setLegend(new ColorbarLegend(surface,
                chart.getView().getAxe().getLayout().getZTickProvider(),
```

```
                                chart.getView().getAxe().getLayout().getZTickRenderer()));
                surface.setLegendDisplayed(true); // opens a colorbar on the right part of the
display
        }
        public Chart getChart(){
                return chart;
        }
        protected Chart chart;
}
```

FilledContoursDemo (Java code - F9 keystroke to execute)

```
import java.awt.Rectangle;

import org.jzy3d.chart.Chart;
import org.jzy3d.colors.Color;
import org.jzy3d.colors.ColorMapper;
import org.jzy3d.colors.colormaps.ColorMapRainbow;
import org.jzy3d.contour.DefaultContourColoringPolicy;
import org.jzy3d.contour.MapperContourPictureGenerator;
import org.jzy3d.demos.AbstractDemo;
import org.jzy3d.demos.IDemo;
import org.jzy3d.demos.Launcher;
import org.jzy3d.factories.JzyFactories;
import org.jzy3d.maths.Range;
import org.jzy3d.plot3d.builder.Builder;
import org.jzy3d.plot3d.builder.Mapper;
import org.jzy3d.plot3d.builder.concrete.OrthonormalGrid;
import org.jzy3d.plot3d.primitives.Shape;
import org.jzy3d.plot3d.primitives.axes.AxeFactory;
import org.jzy3d.plot3d.primitives.axes.ContourAxeBox;
import org.jzy3d.plot3d.primitives.axes.IAxe;
import org.jzy3d.plot3d.rendering.canvas.Quality;
import org.jzy3d.plot3d.rendering.legends.colorbars.ColorbarLegend;
```

```
import org.jzy3d.ui.ChartLauncher;



public class FilledContoursDemo extends AbstractDemo{
	public static void main(String[] args) throws Exception{
		IDemo demo = new FilledContoursDemo();


ChartLauncher.openImagePanel( ((ContourAxeBox)demo.getChart().getView().getAxe()).getConto
urImage(), new Rectangle(600,0,400,400) );
		Launcher.openDemo(demo);
	}


	public FilledContoursDemo(){
		Mapper mapper = new Mapper(){
			public double f(double x, double y) {
				return 10*Math.sin(x/10)*Math.cos(y/20)*x;
			}
		};
		Range xrange = new Range(50,100);   //To get some more detail.
		Range yrange = new Range(50,100);   //To get some more detail.
		int steps   = 50;


		// Create the object to represent the function over the given range.
		final Shape surface = (Shape)Builder.buildOrthonormal(new
OrthonormalGrid(xrange, steps, yrange, steps), mapper);
		ColorMapper myColorMapper=new ColorMapper(new ColorMapRainbow(),
surface.getBounds().getZmin(), surface.getBounds().getZmax(), new Color(1,1,1,.5f));
		surface.setColorMapper(myColorMapper);
		surface.setFaceDisplayed(true);
		surface.setWireframeDisplayed(true);
		surface.setWireframeColor(Color.BLACK);


		// Compute an image of the contour
		MapperContourPictureGenerator contour = new
MapperContourPictureGenerator(mapper, xrange, yrange);


		// Create a chart with contour axe box, and attach the contour picture
```

```
		JzyFactories.axe = new AxeFactory(){
			@Override
			public IAxe getInstance() {
				return new ContourAxeBox(box);
			}
		};
		chart = new Chart(Quality.Advanced);
		ContourAxeBox cab = (ContourAxeBox)chart.getView().getAxe();
		cab.setContourImg( contour.getFilledContourImage(new
DefaultContourColoringPolicy(myColorMapper), 400, 400, 10), xrange, yrange);

		// Add the surface and its colorbar
		chart.addDrawable(surface);
		surface.setLegend(new ColorbarLegend(surface,
				chart.getView().getAxe().getLayout().getZTickProvider(),
				chart.getView().getAxe().getLayout().getZTickRenderer()));
		surface.setLegendDisplayed(true); // opens a colorbar on the right part of the
display
	}

	public Chart getChart(){
		return chart;
	}

	protected Chart chart;
}
```

HeightMapDemo (Java code - F9 keystroke to execute)

```
import java.awt.Rectangle;
import org.jzy3d.chart.Chart;
import org.jzy3d.colors.Color;
```

```
import org.jzy3d.colors.ColorMapper;
import org.jzy3d.colors.colormaps.ColorMapRainbow;
import org.jzy3d.contour.DefaultContourColoringPolicy;
import org.jzy3d.contour.MapperContourPictureGenerator;
import org.jzy3d.demos.AbstractDemo;
import org.jzy3d.demos.IDemo;
import org.jzy3d.demos.Launcher;
import org.jzy3d.factories.JzyFactories;
import org.jzy3d.maths.Range;
import org.jzy3d.plot3d.builder.Builder;
import org.jzy3d.plot3d.builder.Mapper;
import org.jzy3d.plot3d.builder.concrete.OrthonormalGrid;
import org.jzy3d.plot3d.primitives.Shape;
import org.jzy3d.plot3d.primitives.axes.AxeFactory;
import org.jzy3d.plot3d.primitives.axes.ContourAxeBox;
import org.jzy3d.plot3d.primitives.axes.IAxe;
import org.jzy3d.plot3d.primitives.axes.layout.providers.RegularTickProvider;
import org.jzy3d.plot3d.rendering.canvas.Quality;
import org.jzy3d.plot3d.rendering.legends.colorbars.ColorbarLegend;
import org.jzy3d.ui.ChartLauncher;


public class HeightMapDemo extends AbstractDemo{
	public static void main(String[] args) throws Exception{
		IDemo demo = new HeightMapDemo();


ChartLauncher.openImagePanel( ((ContourAxeBox)demo.getChart().getView().getAxe()).getConto
urImage(), new Rectangle(600,0,400,400) );
		Launcher.openDemo(demo);
	}

	public HeightMapDemo(){
		Mapper mapper = new Mapper(){
			public double f(double x, double y) {
				return 10*Math.sin(x/10)*Math.cos(y/20)*x;
			}
```

```
};
Range xrange = new Range(50,100);   //To get some more detail.
Range yrange = new Range(50,100);   //To get some more detail.
int steps   = 50;

// Create the object to represent the function over the given range.
final Shape surface = (Shape)Builder.buildOrthonormal(new
OrthonormalGrid(xrange, steps, yrange, steps), mapper);
ColorMapper myColorMapper=new ColorMapper(new ColorMapRainbow(),
surface.getBounds().getZmin(), surface.getBounds().getZmax(), new Color(1,1,1,.5f));
surface.setColorMapper(myColorMapper);
surface.setFaceDisplayed(true);
surface.setWireframeDisplayed(true);
surface.setWireframeColor(Color.BLACK);

// Compute an image of the contour
MapperContourPictureGenerator contour = new
MapperContourPictureGenerator(mapper, xrange, yrange);

// Create a chart with contour axe box, and attach the contour picture
JzyFactories.axe = new AxeFactory(){
        @Override
        public IAxe getInstance() {
                return new ContourAxeBox(box);
        }
};
chart = new Chart(Quality.Intermediate);
ContourAxeBox cab = (ContourAxeBox)chart.getView().getAxe();
cab.setContourImg( contour.getHeightMap(new
DefaultContourColoringPolicy(myColorMapper), 400, 400, 10), xrange, yrange);

// Add the surface and its colorbar
chart.addDrawable(surface);
surface.setLegend(new ColorbarLegend(surface,
                new RegularTickProvider(10),
                chart.getView().getAxe().getLayout().getZTickRenderer()));
surface.setLegendDisplayed(true); // opens a colorbar on the right part of the
```

```
display
        }
        public Chart getChart(){
                return chart;
        }
        protected Chart chart;
}
```

UserChoosenContoursDemo (Java code - F9 keystroke to execute)

```
import java.awt.Rectangle;
import java.awt.image.BufferedImage;

import org.jzy3d.chart.Chart;
import org.jzy3d.colors.Color;
import org.jzy3d.colors.ColorMapper;
import org.jzy3d.colors.colormaps.ColorMapRainbow;
import org.jzy3d.contour.DefaultContourColoringPolicy;
import org.jzy3d.contour.MapperContourPictureGenerator;
import org.jzy3d.demos.AbstractDemo;
import org.jzy3d.demos.IDemo;
import org.jzy3d.demos.Launcher;
import org.jzy3d.factories.JzyFactories;
import org.jzy3d.maths.Range;
import org.jzy3d.plot3d.builder.Builder;
import org.jzy3d.plot3d.builder.Mapper;
import org.jzy3d.plot3d.builder.concrete.OrthonormalGrid;
import org.jzy3d.plot3d.primitives.Shape;
import org.jzy3d.plot3d.primitives.axes.AxeFactory;
import org.jzy3d.plot3d.primitives.axes.ContourAxeBox;
import org.jzy3d.plot3d.primitives.axes.IAxe;
import org.jzy3d.plot3d.rendering.canvas.Quality;
```

```
import org.jzy3d.plot3d.rendering.legends.colorbars.ColorbarLegend;
import org.jzy3d.ui.ChartLauncher;


public class UserChosenContoursDemo extends AbstractDemo{
	public static void main(String[] args) throws Exception{
		IDemo demo = new UserChosenContoursDemo();

ChartLauncher.openImagePanel( ((ContourAxeBox)demo.getChart().getView().getAxe()).getConto
urImage(), new Rectangle(600,0,400,400) );
		Launcher.openDemo(demo);
	}

	public UserChosenContoursDemo(){
		Mapper mapper = new Mapper(){
			public double f(double x, double y) {
				return 10*Math.sin(x/10)*Math.cos(y/20)*x;
			}
		};
		Range xrange = new Range(50,100);
		Range yrange = new Range(50,100);
		int steps   = 50;

		// Create the object to represent the function over the given range.
		final Shape surface = (Shape)Builder.buildOrthonormal(new
OrthonormalGrid(xrange, steps, yrange, steps), mapper);
		ColorMapper myColorMapper=new ColorMapper(new ColorMapRainbow(),
surface.getBounds().getZmin(), surface.getBounds().getZmax(), new Color(1,1,1,.5f));
		surface.setColorMapper(myColorMapper);
		surface.setFaceDisplayed(true);
		surface.setWireframeDisplayed(true);
		surface.setWireframeColor(Color.BLACK);

		// Compute an image of the contour
		MapperContourPictureGenerator contour = new
MapperContourPictureGenerator(mapper, xrange, yrange);
```

```
		// Create a chart with contour axe box, and attach the contour picture
		JzyFactories.axe = new AxeFactory(){
			@Override
			public IAxe getInstance() {
				return new ContourAxeBox(box);
			}
		};
		chart = new Chart(Quality.Advanced);
		ContourAxeBox cab = (ContourAxeBox)chart.getView().getAxe();

		//Define the array with the heights at which we want a contour line drawn.
Numbers must be ordered from smaller to bigger.
		double sortedContourLevels[]={-500.0,-200.0,0.0, 100.0, 300.0, 400.0};

		//Compute the user-defined contours.
		BufferedImage img = contour.getContourImage(new
DefaultContourColoringPolicy(myColorMapper), 400, 400, sortedContourLevels);
		cab.setContourImg(img, xrange, yrange);

		// Add the surface and its colorbar
		chart.addDrawable(surface);
		surface.setLegend(new ColorbarLegend(surface,
				chart.getView().getAxe().getLayout().getZTickProvider(),
				chart.getView().getAxe().getLayout().getZTickRenderer()));
		surface.setLegendDisplayed(true); // opens a colorbar on the right part of the
display
	}

	public Chart getChart(){
		return chart;
	}

	protected Chart chart;
}
```

Exploiting Java Libraries from GroovyLab

Although, GroovyLab has not currently extensive documentation, the user can still work effectively using the powerful Java libraries that GroovyLab exploits. The philosophy is:

- Convert the GroovySci *Matrix* or the *double[][]* array type to the native representations of the preferrable Java library.
- Thereafter, we perform the desired numerical analysis using the routines of the library, and finally.
- Convert back any desired results to the familiar GroovySci *Matrix* or *double[][]* arrays.

Some useful conversion routines, that are imported by default by GroovyLab are:

```
// JBLAS conversion routines

// convert JBLAS DoubleMatrix to Matrix
public static Matrix JBLAS2Matrix(org.jblas.DoubleMatrix dm)

// convert JBLAS DoubleMatrix to Double Array
public static double[][] JBLAS2DoubleArray(org.jblas.DoubleMatrix dm)

// convert Matrix to JBLAS DoubleMatrix
public static org.jblas.DoubleMatrix  Matrix2JBLAS(Matrix dm)

//// convert double [][] to JBLAS DoubleMatrix
public static org.jblas.DoubleMatrix  Matrix2JBLAS(double [][] dm)


// EJML conversion routines

// convert EJML DenseMatrix64F to Matrix
public static Matrix EJML2Matrix(org.ejml.data.DenseMatrix64F  dm)


// convert EJML  DenseMatrix64F to Double Array
public static double[][] EJML2DoubleArray(org.ejml.data.DenseMatrix64F  dm)

// convert Matrix to EJML DenseMatrix64F
```

```
public static org.ejml.data.DenseMatrix64F   Matrix2EJML(Matrix dm)

//  convert double [][] to JBLAS DoubleMatrix
public static org.ejml.data.DenseMatrix64F  DoubleArray2EJML(double [][] dm)


// MTJ conversion routines

// convert MTJ DenseMatrix to Matrix
public static Matrix MTJ2Matrix(no.uib.cipr.matrix.DenseMatrix   dm)

// convert MTJ DenseMatrix to Double Array
public static double[][] MTJ2DoubleArray(no.uib.cipr.matrix.DenseMatrix  dm)

// convert Matrix to MTJ DenseMatrix
public static no.uib.cipr.matrix.DenseMatrix   Matrix2MTJ(Matrix dm)


// convert double [][] to MTJ DenseMatrix
public static no.uib.cipr.matrix.DenseMatrix  DoubleArray2MTJ(double [][] dm)



// Apache Commons conversion routines

// convert Apache Commons to Matrix
public static Matrix AC2Matrix(org.apache.commons.math3.linear.Array2DRowRealMatrix
dm)

// convert Apache Commons to Double Array
public static double[][]
AC2DoubleArray(org.apache.commons.math3.linear.Array2DRowRealMatrix  dm)

// convert Matrix to Apache Commons
public static org.apache.commons.math3.linear.Array2DRowRealMatrix   Matrix2AC(Matrix
dm)
```

```
  // convert double [][] to Apache Commons
    public static org.apache.commons.math3.linear.Array2DRowRealMatrix
DoubleArray2AC(double [][] dm)
```

EJML Example

For example, we demonstrate how we can use the EJML library to solve a linear system:

```
c = 3.4
A = [[3.4, -0.2, 0.02], [0.34, 5.6, -1.23], [c, -1.2*c, -1.23]] as double[][]    // construct a double []
[]

// use the EJML library to compute the eigenvalues

Ae = DoubleArray2EJML(A)   // convert to the EJML matrix type (i.e.
org.ejml.data.DenseMatrix64F)

// call the library specific routine to compute the solve a system

B = [c, -1.2*c,  -0.1] as double [][]   // the right-hand side part

Be = DoubleArray2EJML(B)  // convert to EJML format

// call the EJML solver, here from the "Libraries" menu, option "EJML routines" we can get
valuable help
// here we convert DenseMatrices to SimpleMatrices which are more convienient
sAe = new org.ejml.simple.SimpleMatrix(Ae)
sBe = new org.ejml.simple.SimpleMatrix(Be)
sXe = sAe.solve(sBe)  // solve the system

dsXe = sXe.matrix   // get the DenseMatrix wrapped in SimpleMatrix
x = EJML2DoubleArray(dsXe)    // convert to the convenient double [][] array

shouldBeZero = A*x-B  // verify that it is zero
```

Fast Computations with the JBLAS library

JBLAS is the fastest Java library (as far as I can know) for linear algebra. To obtain speed it utilizes native BLAS routines. The project page is: ***http://mikiobraun.github.io/jblas/***

GroovyLab automatically installs the relevant staff, therefore it is easy to perform fast computations using JBLAS. We give some examples.

JBLAS operations supported with the Matrix class

The **Matrix** class supports some JBLAS native operations, that are significantly faster than Java based implementations (e.g. about 3 to 7 times, depending on the operation).

Therefore we can use the fast JBLAS eigendecomposition for a **Matrix** object as:

```
A = rand(20)
eigdecomp = A.jblas_eigenvectors()
```

The JBLAS routines are displayed below (is the relevant part of the **Matrix** implementation:

```
   // Computes the eigenvalues of the matrix.using JBLAS
public ComplexDoubleMatrix jblas_eigenvalues() {
   DoubleMatrix dM = new DoubleMatrix(this.getArray());
   return org.jblas.Eigen.eigenvalues(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  ComplexDoubleMatrix[]  jblas_eigenvectors() {
   DoubleMatrix dM = new DoubleMatrix(this.getArray());
   return org.jblas.Eigen.eigenvectors(dM);
}

// Compute the eigenvalues for a symmetric matrix.
```

```
public  DoubleMatrix  jblas_symmetricEigenvalues() {
   DoubleMatrix dM = new DoubleMatrix(this.getArray());
   return org.jblas.Eigen.symmetricEigenvalues(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  DoubleMatrix []  jblas_symmetricEigenvectors() {
   DoubleMatrix dM = new DoubleMatrix(this.getArray());
   return org.jblas.Eigen.symmetricEigenvectors(dM);
}

//  Computes generalized eigenvalues of the problem A x = L B x.
// @param A symmetric Matrix A (A is this). 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  DoubleMatrix jblas_symmetricGeneralizedEigenvalues( double [][] B) {
   return org.jblas.Eigen.symmetricGeneralizedEigenvalues(new DoubleMatrix(this.getArray()),
new DoubleMatrix(B));
}

   /**
    * Solve a general problem A x = L B x.
    *
    * @param A symmetric matrix A ( A is 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.
    */
public DoubleMatrix [] jblas_symmetricGeneralizedEigenvectors( double [][] B) {
   return org.jblas.Eigen.symmetricGeneralizedEigenvectors(new DoubleMatrix(this.getArray()),
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 DoubleMatrix  jblas_cholesky() {
  return org.jblas.Decompose.cholesky(new DoubleMatrix(this.getArray()));
}

/** Solves the linear equation A*X = B.  ( A is this) */
public DoubleMatrix jblas_solve( double [][] B) {
   return org.jblas.Solve.solve(new DoubleMatrix(this.getArray()),  new DoubleMatrix(B));
}

/** Solves the linear equation A*X = B for symmetric A.  ( A is this) */
public DoubleMatrix jblas_solveSymmetric(double [][] B) {
   return org.jblas.Solve.solveSymmetric(new DoubleMatrix(this.getArray()),  new
DoubleMatrix(B));
}

/** Solves the linear equation A*X = B for symmetric and positive definite A. ( A is this ) */
public DoubleMatrix jblas_solvePositive(double [][] B) {
   return org.jblas.Solve.solvePositive(new DoubleMatrix(this.getArray()),  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'
    */

public DoubleMatrix []  jblas_fullSVD( ) {
   return org.jblas.Singular.fullSVD(new DoubleMatrix(this.getArray()));
```

```
}


   /**
    * 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.
    *
    * @param A
    * @return A DoubleMatrix[3] array of U, S, V such that A = U * diag(S) * V'
    */

public  DoubleMatrix []  jblas_sparseSVD( ) {
   return org.jblas.Singular.sparseSVD(new DoubleMatrix(this.getArray()));
}


public ComplexDoubleMatrix []  jblas_sparseSVD( double [][] Aimag) {
   return org.jblas.Singular.sparseSVD(
          new ComplexDoubleMatrix(new DoubleMatrix(this.getArray()),  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  DoubleMatrix jblas_SPDValues() {
   return  org.jblas.Singular.SVDValues(new DoubleMatrix(this.getArray()));
}

   /**
```

```
    * 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  DoubleMatrix  jblas_SPDValues( double [][]Aimag) {
   return  org.jblas.Singular.SVDValues(
         new ComplexDoubleMatrix(new DoubleMatrix(this.getArray()), new DoubleMatrix(Aimag)));
}
```

Matrix multiplication with JBLAS

The following script demonstrates the difference in performance between Native BLAS and Java for matrix multiplication.

```
// Demonstrate the difference in performance between Native BLAS and Java
// for matrix multiplication

 import  org.jblas.*
 n = 1000
 x = DoubleMatrix.randn(n, n)
 y = DoubleMatrix.randn(n, n)
 z = DoubleMatrix.randn(n, n)

 println("Multiplying DOUBLE matrices of size "+ n)

 tic()
 SimpleBlas.gemm(1.0, x, y, 0.0, z)
 tm = toc()

// test with Java multiplication
```

```
xm = rand(n)
tic()
xmxm =xm*xm
tmJ = toc()

println("Time Native = "+tm+", time Java = "+tmJ)
```

Switching of the GroovySci Matrix class to use JBLAS

We can explore the metaprogramming facilities of the Groovy language, in order to dynamically bind the code of many important operations of the **Matrix** class, with the JBLAS implementations.

Therefore, using native implementations offered by JBLAS, we can obtain significant speedup relative to the Java implementations (about 4 to 8 times speedup).

For example, here is how we can reprogram the *matrix multiplication* routine of the *Matrix* class, in order to use JBLAS .

```
// reimplement Matrix-Matrix  multiplication using JBLAS
groovySci.math.array.Matrix.metaClass.multiply = {
  groovySci.math.array.Matrix m ->   // the input Matrix

 // transform the input matrix to the JBLAS representation
    dm =  new org.jblas.DoubleMatrix(m.toDoubleArray())
 // transform the receiver to the JBLAS representation
    dmthis = new org.jblas.DoubleMatrix(delegate.toDoubleArray())
 // fast multiply using JBLAS Native BLAS
    mulRes = dmthis.mmul(dm)

 // return back result as a double [][] array
   groovySci.math.array.JBLASUtils.JBLASDoubleMatrixToDouble2D(mulRes)
}
```

After executing the former script, we can test the speedup. Here, we note that JBLAS obtains significant speed for Linux and Win32 platforms (and MacOS I suppose but I can't test) but not for Win64, where the speed is similar to the Java implementation. Here is a test code:

```
x = rand(2000, 2000) // a large 2000X2000 matrix

tic()
y = x*x  // multiply with JBLAS Native BLAS
tmJBLAS = toc()

xx=Rand(2000, 2000)  // a large 2000X2000 double[][] array
tic()
yy=xx*xx  // multiply with Java
tmJ = toc()

println("time for matrix multiplication using Native BLAS = "+tmJBLAS+", time with Java = "+tmJ)
```

The results for my Linux based PC is:

time for matrix multiplication using Native BLAS = 1.956, time with Java = 10.575

Eigendecomposition with JBLAS

Similarly we can utilize JBLAS for more complex tasks as eigendecomposition

```
// compute the eigenvalues of a general matrix using JBLAS
groovySci.math.array.Matrix.metaClass.jblasEigenValues = {
        // transform the receiver to the JBLAS representation
        dmthis = new org.jblas.DoubleMatrix(delegate.toDoubleArray())
        org.jblas.Eigen.eigenvalues(dmthis)
}
```

To test our routine:

```
xs = rand(4,4)
xeigs = xs.jblasEigenValues()
```

Similarly for eigenvectors:

```
// compute the eigenvectors of a general matrix using JBLAS
//   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.
groovySci.math.array.Matrix.metaClass.jblasEigenVectors = {
        // transform the receiver to the JBLAS representation
        dmthis = new org.jblas.DoubleMatrix(delegate.toDoubleArray())
        org.jblas.Eigen.eigenvectors(dmthis)
}
```

And, we can test:

```
xs2 = rand(3,3)
xeigvecs = xs2.jblasEigenVectors()
```

Similarly for symmetric matrices.

```
//  Compute the eigenvalues for a symmetric matrix.
groovySci.math.array.Matrix.metaClass.jblas_symmetricEigenvalues = {
        // transform the receiver to the JBLAS representation
        dmthis = new org.jblas.DoubleMatrix(delegate.toDoubleArray())
        org.jblas.Eigen.symmetricEigenvalues(dmthis)
}

//  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.
groovySci.math.array.Matrix.metaClass.jblas_symmetricEigenvectors = {
        // transform the receiver to the JBLAS representation
        dmthis = new org.jblas.DoubleMatrix(delegate.toDoubleArray())
```

```
        org.jblas.Eigen.symmetricEigenvectors(dmthis)
}
```

JBLAS also offers a routine for fast Cholesky Decomposition, which can be wrapped as:

```
//  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
 groovySci.math.array.Matrix.metaClass.jblas_cholesky = {
        // transform the receiver to the JBLAS representation
        dmthis = new org.jblas.DoubleMatrix(delegate.toDoubleArray())
        org.jblas.Decompose.cholesky(dmthis)
 }
```

Solution of Linear Systems with JBLAS

Similarly we can use the fast LAPACK based routines for solving linear systems:

```
/** Solves the linear equation A*X = B. */
groovySci.math.array.Matrix.metaClass.jblas_solve = {
  groovySci.math.array.Matrix B ->   // input is the B Matrix, the receiver is A

        // transform the input Matrix B to the JBLAS representation
        Bm =  new org.jblas.DoubleMatrix(B.toDoubleArray())
        // transform the receiver to the JBLAS representation
        dmthis = new org.jblas.DoubleMatrix(delegate.toDoubleArray())
        // return the solution as a convenient GroovySci Matrix

groovySci.math.array.JBLASUtils.JBLASDoubleMatrixToDouble2D(org.jblas.Solve.solve(dmthis, Bm))
        //
groovySci.math.array.JBLASUtils.JBLASDoubleMatrixToMatrix(org.jblas.Solve.solve(dmthis, Bm))  // similar
}
```

and an example:

```
A = rand(3,3)
B = rand(3,3)
x = A.jblas_solve(B)
shouldBeZero = A*x-B
```

FFT With Numerical Recipes Routines

The Numerical Recipes is a classic book, an excellent reference for any researcher and engineer in the field of numerical computation. GroovyLab present a Matlab-like interface to many routines from Numerical Recipes. Numerical Recipe routines are efficient and perhaps even more important they have extensive description in the book. We present some example code based on NR.

FFT with Numerical Recipes based routine

```
closeAll()
 log2N = 12
 N = 2 ** log2N
 t = linspace(0, 1, N)
 dx = t[0,1]-t[0,0]
 SF = 1/dx
 NF = 0.5*SF

 F1  = NF/20

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

realffts = new double[N]  // to accumulate real parts of FFTs
imffts = new double[N]  // to accumulate imaginary parts of FFTs

// Numerical Recipes FFT
NR.NRFFT.four1S(x1.getv(), realffts, imffts)   // perform the FFT
```

```
figure(1);  subplot(2,1,1); plot(realffts, "fft")
subplot(2, 1, 2); plot(x1, "Signal");
```

One-Dimensional FFT example

```
NN=32; NN2=NN+NN
data = new double[NN2]
dcmp = new double[NN2]

  println "TEST 1"
        println "h(t) = real-valued even function"
        println "h(n) = h(N-n) and real"
          // construct the function
        for (int i=0;i<NN2-1;i+=2) {
      data[i]=1.0/((( (i-NN)*(i-NN) )/NN)+1.0)
      data[i+1]=0.0
    }

        cpdata = data.copy()  // copy data since FFT is done in-place

        isign=1
        NR.NRFFT.four1(cpdata, isign);
        figure(1); subplot(2,1,1);     plot(data, Color.BLUE,  "Real-Valued even function");
        subplot(2,1,2); plot(cpdata, Color.RED, "H(n) = H(N-n) and Real")

// print the smallest FFT coefficients up to 5
   NR.NRFFT.printft(cpdata, 5)



        println "TEST 2"
        println "h(t) = imaginary-valued even-function"
        println "h(n) = h(N-n) and imaginary"
        for (int i=0; i<NN2-1; i+=2) {
```

```
data[i+1]=1.0/((( (i-NN)*(i-NN) )/NN)+1.0)
data[i]=0.0;
}

cpdata = data.copy()  // copy data since FFT is done in-place

isign=1
NR.NRFFT.four1(cpdata, isign);
figure(2); subplot(2,1,1);     plot(data, Color.BLUE,  "Imaginary-Valued even function");
subplot(2,1,2); plot(cpdata, Color.RED, "H(n) = H(N-n) ad Imaginary")

println "TEST 3"
println "h(t) = real-valued odd-function"
println "h(n) = -h(N-n) and imaginary"
for (int i=0; i<NN2-1; i+=2) {
    data[i]= (( (i-NN)/NN ))  /  ( (  (i-NN)/NN) *( (i-NN)/NN)  +1.0)
data[i+1]=0.0;
}

// testing simplified interface to four1

realparts = new double[NN]
imparts  = new double[NN]

NR.NRFFT.four1S(data, realparts, imparts)
```

Computing the Power Spectrum of a signal

```
import  static com.nr.NRUtil.SQR
import com.nr.sp.WelchWin
import com.nr.sp.Spectolap
import com.nr.sp.Spectreg

pi = acos(-1.0)

N = 1024
M = (int)(N/2)

spec = new double [M+1]
freq = new double[M+1]

   // Test Spectreg
 welch = new  WelchWin()
 sp=new Spectreg(M)



     // Generate a data set
  t = vlinspace(0, 10, N)
  data = sin(10.34*t)+3.4*cos(5.3*t)

     sp.adddataseg(data.v, welch)
     spec=sp.spectrum()
     freq=sp.frequencies()






figure(1); subplot(2, 1, 1); plot(data, "Data")

subplot(2, 1, 2); plot(freq, spec, "spectrum")
```

Extending the functionality of the GroovySci Matrix using EJML Routines.

EJML (http://code.google.com/p/efficient-java-matrix-library/) is a powerful, well designed Java library for linear algebra. Here, we extend the functionality of the GroovySci *Matrix* class using EJML routines. These routines, sometimes replace the default routines, sometimes are new.

In some cases, we estimate and the obtained speed advantages.

The following script reprograms the metaclass of` *Matrix* to use EJML based implementations.

```
// reimplement Matrix-Matrix  multiplication using EJML
groovySci.math.array.Matrix.metaClass.multiply = {
  groovySci.math.array.Matrix m ->   // the input Matrix

 // transform the input matrix to the EJML representation
    dm =  Matrix2EJML(m)
 // transform the receiver to the EJML  representation
    dmthis = Matrix2EJML(delegate)
 // multiply using EJML
    mulRes = new org.ejml.data.DenseMatrix64F(dmthis.getNumRows(), dm.getNumCols())
   org.ejml.ops.CommonOps.mult(dmthis, dm, mulRes)

 // return back result as a Matrix
  EJML2Matrix(mulRes)
}
```

Example: Comparing EJML and default multiplication: EJML is a little bit faster, even with the   conversions overhead!

```
x = ones(1500)
tic()
y=x*x
tmEJML = toc()
xx = Ones(1500)
```

```
tic()
yy = xx*xx
tmDefault = toc()
```

The following script reimplements *Matrix+Matrix* addition using EJML. It does not gain a speed advantage

```
groovySci.math.array.Matrix.metaClass.plus = {
  groovySci.math.array.Matrix m ->   // the input Matrix

 // transform the input matrix to the EJML DenseMatrix64F representation
    dm =  Matrix2EJML(m)
 // transform the receiver to the EJML  DenseMatrix64F  representation
    dmthis = Matrix2EJML(delegate)
 // multiply using EJML
    plusRes = new org.ejml.data.DenseMatrix64F(dmthis.getNumRows(), dmthis.getNumCols())
   org.ejml.ops.CommonOps.add(dmthis, dm, plusRes)

 // return back result as a Matrix
  EJML2Matrix(plusRes)
}
```

The following script reimplements determinant using EJML. It has similar execution time with JAMA:

```
groovySci.math.array.Matrix.metaClass.det = {
        ->
        // transform the input matrix to the EJML DenseMatrix64F representation
        dmthis = Matrix2EJML(delegate)
        org.ejml.ops.CommonOps.det(dmthis)   // compute and return the determinant
}
```

Also, we reimplement matrix invert using EJML It is faster than JAMA, EJML_time is about 0.7-0.8 * JAMA_time:

```
groovySci.math.array.Matrix.metaClass.inv = {
        ->
        // transform the input matrix to the EJML DenseMatrix64F representation
        dmthis = Matrix2EJML(delegate)
        // create a Matrix to store the inverse
        invMat = new org.ejml.data.DenseMatrix64F(dmthis.getNumRows(),
dmthis.getNumCols())

        org.ejml.ops.CommonOps.invert(dmthis, invMat)   // compute and return the inverse
        EJML2Matrix(invMat)   // return the inverse as a Matrix
}


// computes the Moore-Penrose pseudo-inverse using EJML
groovySci.math.array.Matrix.metaClass.pinv = {
 ->
        // transform the input matrix to the EJML DenseMatrix64F representation
        dmthis = Matrix2EJML(delegate)
        // create a Matrix to store the pseudo-inverse
        pinvMat = new org.ejml.data.DenseMatrix64F(dmthis.getNumCols(),
dmthis.getNumRows())

    // use EJML to compute the pseudo-inverse
    org.ejml.ops.CommonOps.pinv(dmthis, pinvMat)

    EJML2Matrix(pinvMat)   // return the pseudo-inverse as a Matrix
}
```

Transforming GroovyLab scripts to standalone applications

One of the important advantages of GroovyLab is that the scripts can be converted easily to standalone applications, i.e. applications that run on the plain Java platform, without requiring the GroovyLab environment. We illustrate the way to achieve that by means of an example

Suppose that we have the following GroovySci script:

```
def  t = inc(0, 0.01, 10)
def  x = sin(0.23*t)+9.8*cos(1.12*t)
plot(t,x)
```

We want to run that as a **standalone application**.

The first step is to choose from the **Application** menu the **Create stand alone application from the GroovySci script code** option that transforms the Script to standalone application. GroovyLab asks for the name of the class that will wrap the script code, suppose that we use e.g. **plots**. Also a shell script file named with an extension .sh on Unix like systems and .bat on Windows, is created automatically. That file can be used to execute the application directly from the operating system without GroovyLab.

GroovyLab compiles automatically the generated Groovy code that wraps our GroovySci script and produces a corresponding class file on disk.

The second (and final) step is to run our standalone application using the script file, e.g.

sh plots.sh

If we need to run our application (e.g. the **plots** application), to an arbitrary directory, we should be careful to copy the **plots.sh** script, the **plots.class** compiled class, and the **lib** directory also. The later directory is required, since it contains the GroovyLab's libraries and the Groovy .jar file.

Wavelets with GroovyLab

We present one way to work with **Wavelets** in GroovyLab.

The first step is to download the **jWave.jar** toolbox and to install as a GroovyLab toolbox. This is easily accomplished with the **GroovySci toolboxes tab**. In essence this step creates a new GroovyShell, that keeps the variable binding of the previous GroovyShell, while extending the classpath with the new toolboxes.

We can now proceed in using the toolbox with some examples.

Example 1 - Daubechies wavelet with the jWave toolbox

```
// import the relevant material from the jWave toolbox
import  math.transform.jwave.handlers.wavelets.*
import  math.transform.jwave.handlers.*

// create a Daubechies Wavelet
daubWav = new Daub03()

// create a synthetic signal and inject to it some noise
N= 40000
F1 = 23.4; F2 = 0.45; F3= 9.8;
taxis  = inc(0, 5/N, 5-5/N)
sig = 2.5*sin(F1*taxis)+8.9*cos(F2*taxis)-9.8*sin(F3*taxis)
rndSig = rand(1,N)
sigAll = sig+rndSig



// create a FWT object
fwtObj = new FastWaveletTransform(daubWav)

// perform a Fast Wavelet Transform with the Daubechies Wavelet
transfSig = fwtObj.forwardWavelet(sigAll.getv())

// 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")
```

Sparse Matrix class based on MTJ Sparse Matrices.

MTJ offers many sparse matrix classes and associated iterative algorithms for the solution of classical problems.

GroovyLab 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.*



nrows = 10; ncols = 10;
d = new double[nrows][ncols]
d[2][3] = 10
d[4][4] = 44
// create a sparse matrix from the double [][] array
sd = new CCMatrix(d)

d[3][5] =35
sd2 = new CCMatrix(d)

// set an entry using putAt
sd.putAt(2, 1, 21)

// get entries using implicitly getAt
elem2_1 = sd[2,1]
elem2_2 = sd[2,2]
```

```
// test matrix addition
sd1  = sd+sd2
sd
sd1

sd10 = sd*100
sd10

sdd = sd1-sd1
```

Another example of solving sparse systems

```
filename = "C:\\matrixData\\t1"

  // load the sparse matrix stored in triplet format
 A =    loadSparse("L:\\NBProjects\\CSPARSEJ\\CSparseJ\\matrix\\t1")

b = vrand(A.Nrows).getv()

x = solve(A, b)   // solve the system with the CSparse method

Ad = SparseToDoubleArray(A)  // convert to double array

residual = Ad*x - b  // verify: should be near zero



  // convert to an MTJ CCMatrix
 ccms =    CSparseToCCMatrix(A)

 xmtj = solve(ccms, b) // solve with the MTJ based iterative solver

xmtj-x  // verify that the two solutions are equal
```

Sparse Matrix support with the Sparse Class .

ATTENTION: this class is in development, features are not yet stable!!

GroovyLab 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:

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.

s22 = s[2,2]

takes the corresponding element of the matrix.

We can also assign new values, e.g.

s.putAt(1,2, 12)

We can add and multiply two sparse matrices as usual

s2 = s+s

*sMs = s*s*

We can convert the sparse matrix to a double [][] array, as

ds = toDouble(s)

The transpose of a sparse matrix is obtained with the *sparse_t* routine:

ts = sparse_t(s)

We can add and multiply at a sparse matrix a number, e.g.

s10 = s+10

*sm10 = s*10*

We can convert a double [][] to a sparse matrix, e.g.

da = new double[10][20]

da[1][2] = 12

da[2][1] = 21

dasparse = fromDoubleArray(da)

We can negate a sparse matrix:

```
sm = -s
```

We can also add/subtract/multiply to a number a Sparse matrix:

```
sm10a = 10+s
sm10s = 10-s
sm10m = 10*s
```

Applying a closure to all the elements of a sparse matrix can be very convenient:

```
// define a closure cube that performs x^3
 cube= { a -> return (double)a*(double)a*(double)a}

// map the closure to all the elements of the sparse matrix
 scube = s.map(cube)

// display the matrix
 scube.display()
```

Compatibility with Matlab .mat file format

GroovySci provides limited support for loading/saving to Matlab .mat file format. We describe the basic routines by means of examples.

Examples

We can save individual GroovySci variables to .mat files. These variables should be of types double[], double [][], Vec, Matrix. For example:

```
//   save a Matrix
      t = inc(1,1, 1000)
      x = sin(0.12*t)
      matFileName = "testMatrixX"
      save(matFileName, "x")

//    save  a Vector
     vt = vinc(0, 0.01, 10)
     vx = sin(2.3*vt)
     vecFileName = "testVecX"
     save(vecFileName, "vx")
```

We can also save all the variables of types double[], double [][], Vec, Matrix to a .mat file. For example:

```
     // define a Matrix
      t = inc(1,1, 1000)
      x = sin(0.12*t)

     // define a Vector
     vt = vinc(0, 0.01, 10)
     vx = sin(2.3*vt)
```

varsFileName = "testAllVars"

save(varsFileName)

We can load all the saved variables from a .mat file with the load command as the following example illustrates:

load("testAllVars.mat")

Using JTransforms from GroovyLab.

Jtransforms of Piotr Wendykier, is a powerful Java library for Fourier transforms, integrated within the source of GroovyLab. We present here some examples of its use.

Real FFT Example

```
N = 2 **13

dfft = new edu.emory.mathcs.jtransforms.fft.DoubleFFT_1D(N )

data = new double[N]
cpdata = new double[N]
recons = new double[2*N]

// create a signal
for (k in 0..N-1)
  data[k] = 8.9*sin(0.0023*k)+18.9*cos(0.0223*k)+0.02*Math.random()

  // copy it
for (k in 0..N-1)
  cpdata[k] = data[k]

  // perform a real FFT
dfft.realForward(data)
for (k in 0..N-1)
  recons[k] = data[k]



// perform an inverse FFT
dfft.realInverseFull(recons, true)
```

```
figure(1); subplot(3,1,1); plot(cpdata, "original");

validrecons = new double[N]
for (k in 0..N-1)
validrecons[k] = recons[k]

subplot(3,1,2); plot(data, "FFT")

subplot(3,1,3); plot(validrecons, "reconstructed")
```

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 Netwo*rks* 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, 2002

[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] 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

[16] 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

Apache Common Maths

The Apache Common Maths is a powerful library for scientific computing. We adapt here examples in GroovyLab.

The statistics package provides frameworks and implementations for basic Descriptive statistics, frequency distributions, bivariate regression, and t-, chi-square and ANOVA test statistics.

Descriptive Statisticstics

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 double[] arrays 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, n, sum, sum of squares, standard deviation, variance, percentiles, skewness, kurtosis, median	Yes	Yes
SummaryStatistics	min, max, mean, geometric mean, n, sum, sum of squares, standard deviation, variance	No	No

SummaryStatistics can be aggregated using AggregateSummaryStatistics. This class can be used to concurrently gather statistics for multiple datasets as well as for a combined sample including all of the data.

MultivariateSummaryStatistics is similar to SummaryStatistics but handles n-tuple values instead of scalar values. It can also compute the full covariance matrix for the input 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 double[] arrays.

Here are some examples showing how to compute Descriptive statistics.

```
// buffer the imports to execute step-wise
import org.apache.commons.math3.stat.descriptive.*

// Get a DescriptiveStatistics instance
stats = new DescriptiveStatistics()
// Add the data from the array

inputArray = vrand(2000).getv()
for (int i=0; i<inputArray.length; i++)
  stats.addValue(inputArray[i])

// Compute some statistics
mean = stats.getMean()
std = stats.getStandardDeviation()
```

median = stats.getPercentile(50);

Simple Regression

Regression provides ordinary least squares regression with one independent variable estimating the linear model:

y = intercept + slope * x

or

y = 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 requred 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.
- The intercept term may be suppressed by passing false to the SimpleRegression(boolean) constructor. When the hasIntercept property is false, the model is estimated without a constant term andgetIntercept() returns 0.

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 on the assumption that the observations included in the model are drawn from a Bivariate Normal Distribution

Here are some examples.

Estimate a model based on observations added one at a time

Instantiate a regression instance and add data points

// buffer the imports to execute step-wise

*import org.apache.commons.math3.stat.regression.**

```
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)
// Now all statistics are defined

// Compute some statistics based on observations added so far
// displays 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 additional data in statistics.

Estimate a model from a double[][] array of data points

Instantiate a regression o*bject and load dataset*

```
data = [[ 1, 3 ], [2, 5 ], [3, 7 ], [4, 14 ], [5, 11 ]] as double[][]
regression = new SimpleRegression()
regression.addData(data)
```

Estimate regression model based on data

```
// displays 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 additional data in statistics.

Estimate a model from a double[][] array of data points, *excluding* the intercept

Instantiate a regression object and load dataset

```
double[][] data = { { 1, 3 }, {2, 5 }, {3, 7 }, {4, 14 }, {5, 11 }};
SimpleRegression regression = new SimpleRegression(false);
//the argument, false, tells the class not to include a constant
regression.addData(data);
```

Estimate regression model based on data

```
System.out.println(regression.getIntercept());
// displays intercept of regression line, since we have constrained
the constant, 0.0 is returned
System.out.println(regression.getSlope());
// displays slope of regression line
System.out.println(regression.getSlopeStdErr());
// displays slope standard error
System.out.println(regression.getInterceptStdErr() );
// will return Double.NaN, since we constrained the parameter to
zero
```

Caution must be exercised when interpreting the slope when no constant is being estimated. The slope may be biased.

Multiple linear regression

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 Regression, and GLSMultipleLinearRegressionimplements Generalized Least Squares. See the javadoc for these classes for details on the algorithms and forumlas used.

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. See AbstractMultipleLinearRegression#newSampleData(double[],int,int),OLSMultipleLinearRegression#newSampleData(double[], double[][]) andGLSMultipleLinearRegression#newSampleData(double[],double[][],double[][]) for details.

Usage Notes:

- Data are validated when invoking any of the newSample, newX, newY or newCovariance methods andIllegalArgumentException 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

Instantiate an OLS regression object and load a dataset:

```
OLSMultipleLinearRegression regression = new
OLSMultipleLinearRegression();
double[] y = new double[]{11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
double[] x = new double[6][];
x[0] = new double[]{0, 0, 0, 0, 0};
x[1] = new double[]{2.0, 0, 0, 0, 0};
x[2] = new double[]{0, 3.0, 0, 0, 0};
x[3] = new double[]{0, 0, 4.0, 0, 0};
x[4] = new double[]{0, 0, 0, 5.0, 0};
x[5] = new double[]{0, 0, 0, 0, 6.0};
regression.newSample(y, x);
```

Get regression parameters and diagnostics:

```
double[] beta = regression.estimateRegressionParameters();
double[] residuals = regression.estimateResiduals();
double[][] parametersVariance =
regression.estimateRegressionParametersVariance();
double regressandVariance = regression.estimateRegressandVariance();
double rSquared = regression.calculateRSquared();
double sigma = regression.estimateRegressionStandardError();
```

GLS regression

Instantiate a GLS regression object and load a dataset:

```
GLSMultipleLinearRegression regression = new
GLSMultipleLinearRegression();
double[] y = new double[]{11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
double[] x = new double[6][];
x[0] = new double[]{0, 0, 0, 0, 0};
x[1] = new double[]{2.0, 0, 0, 0, 0};
x[2] = new double[]{0, 3.0, 0, 0, 0};
x[3] = new double[]{0, 0, 4.0, 0, 0};
x[4] = new double[]{0, 0, 0, 5.0, 0};
x[5] = new double[]{0, 0, 0, 0, 6.0};
double[][] omega = new double[6][];
omega[0] = new double[]{1.1, 0, 0, 0, 0, 0};
```

```
omega[1] = new double[]{0, 2.2, 0, 0, 0, 0};
omega[2] = new double[]{0, 0, 3.3, 0, 0, 0};
omega[3] = new double[]{0, 0, 0, 4.4, 0, 0};
omega[4] = new double[]{0, 0, 0, 0, 5.5, 0};
omega[5] = new double[]{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. Theorg.apache.commons.math3.stat.ranking package provides rank transformation. RankingAlgorithm defines the interface for ranking. NaturalRanking provides an implementation that has two configuration options.

- Ties strategy deterimines how ties in the source data are handled by the ranking
- NaN strategy determines how NaN values in the source data are handled.

Examples:

```
NaturalRanking ranking = new NaturalRanking(NaNStrategy.MINIMAL,
TiesStrategy.MAXIMUM);
double[] data = { 20, 17, 30, 42.3, 17, 50,
                  Double.NaN, Double.NEGATIVE_INFINITY, 17 };
double[] ranks = ranking.rank(exampleData);
```

results in ranks containing {6, 5, 7, 8, 5, 9, 2, 2, 5}.

```
new
NaturalRanking(NaNStrategy.REMOVED,TiesStrategy.SEQUENTIAL).rank(exampleD
ata);
```

returns {5, 2, 6, 7, 3, 8, 1, 4}.

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. See the NaturalRanking for more examples and TiesStrategy and NaNStrategy for details on these configuration options.

Covariance and correlation

The org.apache.commons.math3.stat.correlation package computes covariances and correlations for pairs of arrays or columns of a matrix. Covariance computes covariances, PearsonsCorrelation provides Pearson's Product-Moment correlation coefficients and SpearmansCorrelation computes Spearman's rank correlation.

Implementation Notes

- Unbiased covariances are given by the formula

  cov(X, Y) = sum [(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 the 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) = sum[(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:

```
new Covariance().covariance(x, y)
```

For non-bias-corrected covariances, use

```
covariance(x, y, false)
```

Covariance matrix

A covariance matrix over the columns of a source matrix data can be computed using

```
new Covariance().computeCovarianceMatrix(data)
```

The i-jth entry of the returned matrix is the unbiased covariance of the ith and jth columns of data. As above, to get non-bias-corrected covariances, use

```
computeCovarianceMatrix(data, false)
```

Pearson's correlation of 2 arrays

To compute the Pearson's product-moment correlation between two double arrays x and y, use:

```
new PearsonsCorrelation().correlation(x, y)
```

Pearson's correlation matrix

A (Pearson's) correlation matrix over the columns of a source matrix data can be computed using

```
new PearsonsCorrelation().computeCorrelationMatrix(data)
```

The i-jth entry of the returned matrix is the Pearson's product-moment correlation between the ith and jth columns of data.

Pearson's correlation significance and standard errors

To compute standard errors and/or significances of correlation coefficients associated with Pearson's correlation coefficients, start by creating a PearsonsCorrelation instance

```
PearsonsCorrelation correlation = new PearsonsCorrelation(data);
```

where data is either a rectangular array or a RealMatrix. Then the matrix of standard errors is

```
correlation.getCorrelationStandardErrors();
```

The formula used to compute the standard error is
$SE_r = ((1 - r^2) / (n - 2))^{1/2}$
where r is the estimated correlation coefficient and n is the number of observations in the source dataset.

p-values for the (2-sided) null hypotheses that elements of a correlation matrix are zero populate the RealMatrix returned by

```
correlation.getCorrelationPValues()
```

getCorrelationPValues().getEntry(i,j) is the probability that a random variable distributed as t_{n-2}takes a value with absolute value greater than or equal to

$|r_{ij}|((n - 2) / (1 - r_{ij}^2))^{1/2}$, where r_{ij} is the estimated correlation between the ith and jth columns of the source array or RealMatrix. This is sometimes referred to as the *significance* of the coefficient.

For example, if data is a RealMatrix with 2 columns and 10 rows, then

```
new PearsonsCorrelation(data).getCorrelationPValues().getEntry(0,1)
```

is the significance of the Pearson's correlation coefficient between the two columns of data. If this value is less than .01, we can say that the correlation between the two columns of data is significant at the 99% level.

Spearman's rank correlation coefficient

To compute the Spearman's rank-moment correlation between two double arrays x and y:

```
new SpearmansCorrelation().correlation(x, y)
```

This is equivalent to

```
RankingAlgorithm ranking = new NaturalRanking();
new PearsonsCorrelation().correlation(ranking.rank(x),
ranking.rank(y))
```

Statistical tests

The org.apache.commons.math3.stat.inference package provides implementations for Student's t, Chi-Square, G Test, One-Way ANOVA, Mann-Whitney U and Wilcoxon signed rank test statistics as well as p-values associated with t-, Chi-Square, G, One-Way ANOVA, Mann-Whitney U and Wilcoxon signed rank tests. The respective test classes are TTest, ChiSquareTest, GTest, OneWayAnova,MannWhitneyUTest, and WilcoxonSignedRankTest. The TestUtils class provides static methods to get test instances or to compute test statistics directly. The examples below all use the static methods in TestUtilsto execute tests. To get test object instances, either use e.g., TestUtils.getTTest() or use the implementation constructors directly, e.g. new TTest().

Implementation Notes

- Both one- and two-sample t-tests are supported. Two sample tests can be either paired or unpaired and the unpaired two-sample tests can be conducted under the assumption of equal subpopulation variances or without this assumption. When equal variances is assumed, a pooled variance estimate is used to compute the t-statistic and the degrees of freedom used in the t-test equals the sum of the sample sizes minus 2. When equal variances is not assumed, the t-statistic uses both sample variances and theWelch-Satterwaite approximation is used to compute the degrees of freedom. Methods to return t-statistics and p-values are provided in each case, as well as boolean-valued methods to perform fixed significance level tests. The names of methods or methods that assume equal subpopulation variances always start with "homoscedastic." Test or test-statistic methods that just start with "t" do not assume equal variances. See the examples below and the API documentation for more details.
- The validity of the p-values returned by the t-test depends on the assumptions of the parametric t-test procedure, as discussed here
- p-values returned by t-, chi-square and Anova tests are exact, based on numerical approximations to the t-, chi-square and F distributions in the distributions package.
- The G test implementation provides two p-values: gTest(expected, observed), which is the tail probability beyond g(expected, observed) in the ChiSquare distribution with degrees of freedom one less than the common length of input arrays and gTestIntrinsic(expected, observed) which is the same tail probability computed using a ChiSquare distribution with one less degeree of freedom.
- p-values returned by t-tests are for two-sided tests and the boolean-valued methods supporting fixed significance level tests assume that the hypotheses are two-sided. One sided tests can be performed by dividing returned p-values (resp. critical values) by 2.
- Degrees of freedom for G- and chi-square tests are integral values, based on the number of observed or expected counts (number of observed counts - 1).

Examples:

One-sample t tests

To compare the mean of a double[] array to a fixed value:

```
double[] observed = {1d, 2d, 3d};
double mu = 2.5d;
System.out.println(TestUtils.t(mu, observed));
```

The code above will display the t-statisitic associated with a one-sample t-test comparing the mean of the observed values against mu.
To compare the mean of a dataset described by a StatisticalSummary to a fixed value:

```
double[] observed ={1d, 2d, 3d};
double mu = 2.5d;
SummaryStatistics sampleStats = new SummaryStatistics();
for (int i = 0; i < observed.length; i++) {
    sampleStats.addValue(observed[i]);
}
System.out.println(TestUtils.t(mu, observed));
```

To compute the p-value associated with the null hypothesis that the mean of a set of values equals a point estimate, against the two-sided alternative that the mean is different from the target value:

```
double[] observed = {1d, 2d, 3d};
double mu = 2.5d;
System.out.println(TestUtils.tTest(mu, observed));
```

The snippet above will display the p-value associated with the null hypothesis that the mean of the population from which the observed values are drawn equals mu.
To perform the test using a fixed significance level, use:

```
TestUtils.tTest(mu, observed, alpha);
```

where 0 < alpha < 0.5 is the significance level of the test. The boolean value returned will be true iff the null hypothesis can be rejected with confidence 1 - alpha. To test, for example at the 95% level of confidence, use alpha = 0.05

Two-Sample t-tests

Example 1: Paired test evaluating the null hypothesis that the mean difference between corresponding (paired) elements of the double[] arrays sample1 and sample2 is zero.

To compute the t-statistic:

```
TestUtils.pairedT(sample1, sample2);
```

To compute the p-value:

```
TestUtils.pairedTTest(sample1, sample2);
```

To perform a fixed significance level test with alpha = .05:

```
TestUtils.pairedTTest(sample1, sample2, .05);
```

The last example will return true iff the p-value returned by TestUtils.pairedTTest(sample1, sample2) is less than .05
Example 2: unpaired, two-sided, two-sample t-test using StatisticalSummary instances, without assuming that subpopulation variances are equal.

First create the StatisticalSummary instances. Both DescriptiveStatistics andSummaryStatistics implement this interface. Assume that summary1 and summary2 areSummaryStatistics instances, each of which has had at least 2

values added to the (virtual) dataset that it describes. The sample sizes do not have to be the same -- all that is required is that both samples have at least 2 elements.

Note: The SummaryStatistics class does not store the dataset that it describes in memory, but it does compute all statistics necessary to perform t-tests, so this method can be used to conduct t-tests with very large samples. One-sample tests can also be performed this way. (See Descriptive statistics for details on the SummaryStatistics class.)

To compute the t-statistic:

```
TestUtils.t(summary1, summary2);
```

To compute the p-value:

```
TestUtils.tTest(sample1, sample2);
```

To perform a fixed significance level test with alpha = .05:

```
TestUtils.tTest(sample1, sample2, .05);
```

In each case above, the test does not assume that the subpopulation variances are equal. To perform the tests under this assumption, replace "t" at the beginning of the method name with "homoscedasticT"

Chi-square tests

To compute a chi-square statistic measuring the agreement between a long[] array of observed counts and a double[] array of expected counts, use:

```
long[] observed = {10, 9, 11};
double[] expected = {10.1, 9.8, 10.3};
System.out.println(TestUtils.chiSquare(expected, observed));
```

the value displayed will be sum((expected[i] - observed[i])^2 / expected[i])
To get the p-value associated with the null hypothesis that observed conforms to expected use:

```
TestUtils.chiSquareTest(expected, observed);
```

To test the null hypothesis that observed conforms to expected with alpha siginficance level (equiv.100 * (1-alpha)% confidence) where 0 < alpha < 1 use:

```
TestUtils.chiSquareTest(expected, observed, alpha);
```

The boolean value returned will be true iff the null hypothesis can be rejected with confidence 1 - alpha.
To compute a chi-square statistic statistic associated with a chi-square test of independence based on a two-dimensional (long[][]) counts array viewed as a two-way table, use:

```
TestUtils.chiSquareTest(counts);
```

The rows of the 2-way table are count[0], ... , count[count.length - 1].

The chi-square statistic returned is sum((counts[i][j] - expected[i][j])^2/expected[i][j])where the sum is taken over all table entries and expected[i][j] is the product of the row and column sums at row i, column j divided by the total count.
To compute the p-value associated with the null hypothesis that the classifications represented by the counts in the columns of the input 2-way table are independent of the rows, use:

```
TestUtils.chiSquareTest(counts);
```

To perform a chi-square test of independence with alpha siginficance level (equiv. 100 * (1-alpha)%confidence) where 0 < alpha < 1 use:

```
TestUtils.chiSquareTest(counts, alpha);
```

The boolean value returned will be true iff the null hypothesis can be rejected with confidence 1 - alpha.

G tests

G tests are an alternative to chi-square tests that are recommended when observed counts are small and / or incidence probabillities for some cells are small. See Ted Dunning's paper, Accurate Methods for the Statistics of Surprise and Coincidence for background and an empirical analysis showing now chi-square statistics can be misldeading in the presence of low incidence probabilities. This paper also derives the formulas used in computing G statistics and the root log likelihood ratio provided by the GTestclass.
To compute a G-test statistic measuring the agreement between a long[] array of observed counts and a double[] array of expected counts, use:

```
double[] expected = new double[]{0.54d, 0.40d, 0.05d, 0.01d};
long[] observed = new long[]{70, 79, 3, 4};
System.out.println(TestUtils.g(expected, observed));
```

the value displayed will be 2 * sum(observed[i]) * log(observed[i]/expected[i])
To get the p-value associated with the null hypothesis that observed conforms to expected use:

```
TestUtils.gTest(expected, observed);
```

To test the null hypothesis that observed conforms to expected with alpha siginficance level (equiv.100 * (1-alpha)% confidence) where 0 < alpha < 1 use:

```
TestUtils.gTest(expected, observed, alpha);
```

The boolean value returned will be true iff the null hypothesis can be rejected with confidence 1 - alpha.
To evaluate the hypothesis that two sets of counts come from the same underlying distribution, use long[] arrays for the counts and gDataSetsComparison for the test statistic

```
long[] obs1 = new long[]{268, 199, 42};
long[] obs2 = new long[]{807, 759, 184};
System.out.println(TestUtils.gDataSetsComparison(obs1, obs2)); // G 
statistic
```

```
System.out.println(TestUtils.gTestDataSetsComparison(obs1, obs2));
// p-value
```

For 2 x 2 designs, the rootLogLikelihoodRaio method computes the signed root log likelihood ratio.For example, suppose that for two events A and B, the observed count of AB (both occurring) is 5, not A and B (B without A) is 1995, A not B is 0; and neither A nor B is 10000. Then

```
new GTest().rootLogLikelihoodRatio(5, 1995, 0, 100000);
```

returns the root log likelihood associated with the null hypothesis that A and B are independent.

One-Way Anova tests

```
double[] classA =
   {93.0, 103.0, 95.0, 101.0, 91.0, 105.0, 96.0, 94.0, 101.0 };
double[] classB =
   {99.0, 92.0, 102.0, 100.0, 102.0, 89.0 };
double[] classC =
   {110.0, 115.0, 111.0, 117.0, 128.0, 117.0 };
List classes = new ArrayList();
classes.add(classA);
classes.add(classB);
classes.add(classC);
```

Then you can compute ANOVA F- or p-values associated with the null hypothesis that the class means are all the same using a OneWayAnova instance or TestUtils methods:

```
double fStatistic = TestUtils.oneWayAnovaFValue(classes); // F-value
double pValue = TestUtils.oneWayAnovaPValue(classes);     // P-value
```

To test perform a One-Way Anova test with signficance level set at 0.01 (so the test will, assuming assumptions are met, reject the null hypothesis incorrectly only about one in 100 times), use

```
TestUtils.oneWayAnovaTest(classes, 0.01); // returns a boolean
                                           // true means reject null
hypothesis
```

Numerical Analysis

The analysis package is the parent package for algorithms dealing with real-valued functions of one real variable. It contains dedicated sub-packages providing numerical root-finding, integration, interpolation and differentiation. It also contains a polynomials sub-package that considers polynomials with real coefficients as differentiable real functions.

Functions interfaces are intended to be implemented by user code to represent their domain problems. The algorithms provided by the library will then operate on these function to find their roots, or integrate them, or ... Functions can be multivariate or univariate, real vectorial or matrix valued, and they can be differentiable or not.

Error handling

For user-defined functions, when the method encounters an error during evaluation, users must use their*own* unchecked exceptions. The following example shows the recommended way to do that, using root solving as the example (the same construct should be used for ODE integrators or for optimizations).

```
private static class LocalException extends RuntimeException {
  // the x value that caused the problem
  private final double x;
  public LocalException(double x) {
    this.x = x;
  }
  public double getX() {
    return x;
  }
}
private static class MyFunction implements UnivariateFunction {
  public double value(double x) {
    double y = hugeFormula(x);
    if (somethingBadHappens) {
      throw new LocalException(x);
    }
    return y;
  }
}
public void compute() {
  try {
    solver.solve(maxEval, new MyFunction(a, b, c), min, max);
  } catch (LocalException le) {
    // retrieve the x value
  }
}
```

As shown in this example the exception is really something local to user code and there is a guarantee Apache Commons Math will not mess with it. The user is safe.

Root-finding

UnivariateSolver, UnivariateDifferentiableSolver and PolynomialSolver provide means to find roots of univariate real-valued functions, differentiable univariate real-valued functions, and polynomial functions respectively. A root is the value where the function takes the value 0. Commons-Math includes implementations of the several root-finding algorithms:

Root solvers				
Name	Function type	Convergence	Needs initial bracketing	Bracket side selection
Bisection	univariate real-valued functions	linear, guaranteed	yes	yes
Brent-Dekker	univariate real-valued functions	super-linear, guaranteed	yes	no
bracketing n^{th} order Brent	univariate real-valued functions	variable order, guaranteed	yes	yes
Illinois Method	univariate real-valued functions	super-linear, guaranteed	yes	yes
Laguerre's Method	polynomial functions	cubic for simple root, linear for multiple root	yes	no
Muller's Method using bracketing to deal with real-valued functions	univariate real-valued functions	quadratic close to roots	yes	no
Muller's Method using modulus to deal with real-valued functions	univariate real-valued functions	quadratic close to root	yes	no
Newton-Raphson's Method	differentiable univariate real-valued functions	quadratic, non-guaranteed	no	no
Pegasus Method	univariate real-valued functions	super-linear, guaranteed	yes	yes
Regula Falsi (false position) Method	univariate real-valued functions	linear, guaranteed	yes	yes
Ridder's Method	univariate real-valued	super-linear	yes	no

	functions			
Secant Method	univariate real-valued functions	super-linear, non-guaranteed	yes	no

Some algorithms require that the initial search interval brackets the root (i.e. the function values at interval end points have opposite signs). Some algorithms preserve bracketing throughout computation and allow user to specify which side of the convergence interval to select as the root. It is also possible to force a side selection after a root has been found even for algorithms that do not provide this feature by themselves. This is useful for example in sequential search, for which a new search interval is started after a root has been found in order to find the next root. In this case, user must select a side to ensure his loop is not stuck on one root and always return the same solution without making any progress.

There are numerous non-obvious traps and pitfalls in root finding. First, the usual disclaimers due to the way real world computers calculate values apply. If the computation of the function provides numerical instabilities, for example due to bit cancellation, the root finding algorithms may behave badly and fail to converge or even return bogus values. There will not necessarily be an indication that the computed root is way off the true value. Secondly, the root finding problem itself may be inherently ill-conditioned. There is a "domain of indeterminacy", the interval for which the function has near zero absolute values around the true root, which may be large. Even worse, small problems like roundoff error may cause the function value to "numerically oscillate" between negative and positive values. This may again result in roots way off the true value, without indication. There is not much a generic algorithm can do if ill-conditioned problems are met. A way around this is to transform the problem in order to get a better conditioned function. Proper selection of a root-finding algorithm and its configuration parameters requires knowledge of the analytical properties of the function under analysis and numerical analysis techniques. Users are encouraged to consult a numerical analysis text (or a numerical analyst) when selecting and configuring a solver.

In order to use the root-finding features, first a solver object must be created by calling its constructor, often providing relative and absolute accuracy. Using a solver object, roots of functions are easily found using the solve methods. These methods takes a maximum iteration count maxEval, a function f, and either two domain values, min and max, or a startValue as parameters. If the maximal number of iterations count is exceeded, non-convergence is assumed and a ConvergenceException exception is thrown. A suggested value is 100, which should be plenty, given that a bisection algorithm can't get any more accurate after 52 iterations because of the number of mantissa bits in a double precision floating point number. If a number of ill-conditioned problems is to be solved, this number can be decreased in order to avoid wasting time.Bracketed solvers also take an allowed solution enum parameter to specify which side of the final convergence interval should be selected as the root. It can be ANY_SIDE, LEFT_SIDE, RIGHT_SIDE,BELOW_SIDE or ABOVE_SIDE. Left and right are used to specify the root along the function parameter axis while below and above refer to the function value axis. The solve methods compute a value c such that:

- f(c) = 0.0 (see "function value accuracy")
- min <= c <= max (except for the secant method, which may find a solution outside the interval)

Typical usage:

```
UnivariateFunction function = // some user defined function object
final double relativeAccuracy = 1.0e-12;
final double absoluteAccuracy = 1.0e-8;
final int    maxOrder         = 5;
UnivariateSolver solver   = new
BracketingNthOrderBrentSolver(relativeAccuracy, absoluteAccuracy,
maxOrder);
double c = solver.solve(100, function, 1.0, 5.0,
AllowedSolution.LEFT_SIDE);
```

Force bracketing, by refining a base solution found by a non-bracketing solver:

```
UnivariateFunction function = // some user defined function object
final double relativeAccuracy = 1.0e-12;
final double absoluteAccuracy = 1.0e-8;
UnivariateSolver nonBracketing = new BrentSolver(relativeAccuracy,
absoluteAccuracy);
double baseRoot = nonBracketing.solve(100, function, 1.0, 5.0);
double c = UnivariateSolverUtils.forceSide(100, function,
                                            new
PegasusSolver(relativeAccuracy, absoluteAccuracy),
                                            baseRoot, 1.0, 5.0,
AllowedSolution.LEFT_SIDE);
```

The BrentSolver uses the Brent-Dekker algorithm which is fast and robust. If there are multiple roots in the interval, or there is a large domain of indeterminacy, the algorithm will converge to a random root in the interval without indication that there are problems. Interestingly, the examined text book implementations all disagree in details of the convergence criteria. Also each implementation had problems for one of the test cases, so the expressions had to be fudged further. Don't expect to get exactly the same root values as for other implementations of this algorithm.

The BracketingNthOrderBrentSolver uses an extension of the Brent-Dekker algorithm which uses inverse n^{th} order polynomial interpolation instead of inverse quadratic interpolation, and which allows selection of the side of the convergence interval for result bracketing. This is now the recommended algorithm for most users since it has the largest order, doesn't require derivatives, has guaranteed convergence and allows result bracket selection.

The SecantSolver uses a straightforward secant algorithm which does not bracket the search and therefore does not guarantee convergence. It may be faster than Brent on some well-behaved functions.

The RegulaFalsiSolver is variation of secant preserving bracketing, but then it may be slow, as one end point of the search interval will become fixed after and only the other end point will converge to the root, hence resulting in a search interval size that does not decrease to zero.

The IllinoisSolver and PegasusSolver are well-known variations of regula falsi that fix the problem of stuck end points by slightly weighting one endpoint to balance the interval at next iteration. Pegasus is often faster than Illinois. Pegasus may be the algorithm of choice for selecting a specific side of the convergence interval.

The BisectionSolver is included for completeness and for establishing a fall back in cases of emergency. The algorithm is simple, most likely bug free and guaranteed to converge even in very adverse circumstances which might cause other algorithms to malfunction. The drawback is of course that it is also guaranteed to be slow.

The UnivariateSolver interface exposes many properties to control the convergence of a solver. The accuracy properties are set at solver instance creation and cannot be changed afterwards, there are only getters to retriveve their values, no setters are available.

Property	Purpose
Absolute accuracy	The Absolute Accuracy is (estimated) maximal difference between the computed root and the true root of the function. This is what most people think of as "accuracy" intuitively. The default value is chosen as a sane value for most real world problems, for roots in the range from -100 to +100. For accurate computation of roots near zero, in the range form -0.0001 to +0.0001, the value may be decreased. For computing roots much larger in absolute value than 100, the default absolute accuracy may never be reached because the given relative accuracy is reached first.

Relative accuracy	The Relative Accuracy is the maximal difference between the computed root and the true root, divided by the maximum of the absolute values of the numbers. This accuracy measurement is better suited for numerical calculations with computers, due to the way floating point numbers are represented. The default value is chosen so that algorithms will get a result even for roots with large absolute values, even while it may be impossible to reach the given absolute accuracy.
Function value accuracy	This value is used by some algorithms in order to prevent numerical instabilities. If the function is evaluated to an absolute value smaller than the Function Value Accuracy, the algorithms assume they hit a root and return the value immediately. The default value is a "very small value". If the goal is to get a near zero function value rather than an accurate root, computation may be sped up by setting this value appropriately.

Interpolation

A UnivariateInterpolator is used to find a univariate real-valued function f which for a given set of ordered pairs (x_i,y_i) yields $f(x_i)=y_i$ to the best accuracy possible. The result is provided as an object implementing the UnivariateFunction interface. It can therefore be evaluated at any point, including point not belonging to the original set. Currently, only an interpolator for generating natural cubic splines and a polynomial interpolator are available. There is no interpolator factory, mainly because the interpolation algorithm is more determined by the kind of the interpolated function rather than the set of points to interpolate. There aren't currently any accuracy controls either, as interpolation accuracy is in general determined by the algorithm.

Typical usage:

```
double x[] = { 0.0, 1.0, 2.0 };
double y[] = { 1.0, -1.0, 2.0);
UnivariateInterpolator interpolator = new SplineInterpolator();
UnivariateFunction function = interpolator.interpolate(x, y);
double interpolationX = 0.5;
double interpolatedY = function.evaluate(x);
System.out println("f(" + interpolationX + ") = " + interpolatedY);
```

A natural cubic spline is a function consisting of a polynomial of third degree for each subinterval determined by the x-coordinates of the interpolated points. A function interpolating N value pairs consists of N-1polynomials. The function is continuous, smooth and can be differentiated twice. The second derivative is continuous but not smooth. The x values passed to the interpolator must be ordered in ascending order. It is not valid to evaluate the function for values outside the range $x_0..x_N$.

The polynomial function returned by the Neville's algorithm is a single polynomial guaranteed to pass exactly through the interpolation points. The degree of the polynomial is the number of points minus 1 (i.e. the interpolation polynomial for a three points set will be a quadratic polynomial). Despite the fact the interpolating polynomials is a perfect approximation of a function at interpolation points, it may be a loose approximation between the points. Due to Runge's phenomenom the error can get worse as the degree of the polynomial increases, so adding more points does not always lead to a better interpolation.

Loess (or Lowess) interpolation is a robust interpolation useful for smoothing univariate scaterplots. It has been described by William Cleveland in his 1979 seminal paper Robust Locally Weighted Regression and Smoothing Scatterplots. This kind of interpolation is computationally intensive but robust.

Microsphere interpolation is a robust multidimensional interpolation algorithm. It has been described in William Dudziak's MS thesis.

Hermite interpolation is an interpolation method that can use derivatives in addition to function values at sample points. The HermiteInterpolator class implements this method for vector-valued functions. The

sampling points can have any spacing (there are no requirements for a regular grid) and some points may provide derivatives while others don't provide them (or provide derivatives to a smaller order). Points are added one at a time, as shown in the following example:

```
HermiteInterpolator interpolator = new HermiteInterpolator;
// at x = 0, we provide both value and first derivative
interpolator.addSamplePoint(0.0, new double[] { 1.0 }, new double[] { 2.0 });
// at x = 1, we provide only function value
interpolator.addSamplePoint(1.0, new double[] { 4.0 });
// at x = 2, we provide both value and first derivative
interpolator.addSamplePoint(2.0, new double[] { 5.0 }, new double[] { 2.0 });
// should print "value at x = 0.5: 2.5625"
System.out.println("value at x = 0.5: " + interpolator.value(0.5)[0]);
// should print "derivative at x = 0.5: 3.5"
System.out.println("derivative at x = 0.5: " + interpolator.derivative(0.5)[0]);
// should print "interpolation polynomial: 1 + 2 x + 4 x^2 - 4 x^3 + x^4"
System.out.println("interpolation polynomial: " + interpolator.getPolynomials()[0]);
```

A BivariateGridInterpolator is used to find a bivariate real-valued function f which for a given set of tuples (x_i,y_j,f_{ij}) yields f(x_i,y_j)=f_{ij} to the best accuracy possible. The result is provided as an object implementing the BivariateFunction interface. It can therefore be evaluated at any point, including a point not belonging to the original set. The arrays x_i and y_j must be sorted in increasing order in order to define a two-dimensional grid.

In bicubic interpolation, the interpolation function is a 3rd-degree polynomial of two variables. The coefficients are computed from the function values sampled on a grid, as well as the values of the partial derivatives of the function at those grid points. From two-dimensional data sampled on a grid, theBicubicSplineInterpolator computes a bicubic interpolating function. Prior to computing an interpolating function, the SmoothingPolynomialBicubicSplineInterpolator class performs smoothing of the data by computing the polynomial that best fits each of the one-dimensional curves along each of the coordinate axes.

A TrivariateGridInterpolator is used to find a trivariate real-valued function f which for a given set of tuples (x_i,y_j,z_k, f_{ijk}) yields f(x_i,y_j,z_k)=f_{ijk} to the best accuracy possible. The result is provided as an object implementing the TrivariateFunction interface. It can therefore be evaluated at any point, including a point not belonging to the original set. The arrays x_i, y_j and z_k must be sorted in increasing order in order to define a three-dimensional grid.

In tricubic interpolation, the interpolation function is a 3rd-degree polynomial of three variables. The coefficients are computed from the function values sampled on a grid, as well as the values of the partial derivatives of the function at those grid points. From three-dimensional data sampled on a grid, theTricubicSplineInterpolator computes a tricubic interpolating function.

Integration

A UnivariateIntegrator provides the means to numerically integrate univariate real-valued functions. Commons-Math includes implementations of the following integration algorithms:

- Romberg's method
- Simpson's method
- trapezoid method
- Legendre-Gauss method

Polynomials

The org.apache.commons.math3.analysis.polynomials package provides real coefficients polynomials.

The PolynomialFunction class is the most general one, using traditional coefficients arrays. ThePolynomialsUtils utility class provides static factory methods to build Chebyshev, Hermite, Jacobi, Laguerre and Legendre polynomials. Coefficients are computed using exact fractions so these factory methods can build polynomials up to any degree.

Differentiation

The org.apache.commons.math3.analysis.differentiation package provides a general-purpose differentiation framework.

The core class is DerivativeStructure which holds the value and the differentials of a function. This class handles some arbitrary number of free parameters and arbitrary derivation order. It is used both as the input and the output type for the UnivariateDifferentiableFunction interface. Any differentiable function should implement this interface.

The main idea behind the DerivativeStructure class is that it can be used almost as a number (i.e. it can be added, multiplied, its square root can be extracted or its cosine computed... However, in addition to computed the value itself when doing these computations, the partial derivatives are also computed alongside. This is an extension of what is sometimes called Rall's numbers. This extension is described in Dan Kalman's paper Doubly Recursive Multivariate Automatic Differentiation, Mathematics Magazine, vol. 75, no. 3, June 2002. Rall's numbers only hold the first derivative with respect to one free parameter whereas Dan Kalman's derivative structures hold all partial derivatives up to any specified order, with respect to any number of free parameters. Rall's numbers therefore can be seen as derivative structures for order one derivative and one free parameter, and primitive real numbers can be seen as derivative structures with zero order derivative and no free parameters.

The workflow of computation of a derivatives of an expression y=f(x) is the following one. First we configure an input parameter x of type DerivativeStructure so it will drive the function to compute all derivatives up to order 3 for example. Then we compute y=f(x) normally by passing this parameter to the f function.At the end, we extract from y the value and the derivatives we want. As we have specified 3rd order when we builtx, we can retrieve the derivatives up to 3rd order from y. The following example shows that (the 0 parameter in the DerivativeStructure constructor will be explained in the next paragraph):

```
int params = 1;
int order = 3;
double xRealValue = 2.5;
DerivativeStructure x = new DerivativeStructure(params, order, 0, xRealValue);
DerivativeStructure y = f(x);
System.out.println("y    = " + y.getValue();
System.out.println("y'   = " + y.getPartialDerivative(1);
System.out.println("y''  = " + y.getPartialDerivative(2);
System.out.println("y''' = " + y.getPartialDerivative(3);
```

In fact, there are no notions of *variables* in the framework, so neither x nor y are considered to be variables per se. They are both considered to be *functions* and to depend on implicit free parameters which are represented only by indices in the framework. The x instance above is there considered by the framework to be a function of free parameter p0 at index 0, and as y is computed from x it is the result of a functions composition and is therefore also a function of this p0 free parameter. The p0 is not represented by itself, it is simply defined implicitely by the 0 index above. This index is the third argument in the constructor of the xinstance. What this constructor means is that we built x as a function that depends on one free parameter only (first constructor argument set to 1), that can be differentiated up to order 3 (second constructor argument set to 3), and which correspond to an identity function with respect to implicit free parameter number 0 (third constructor argument set to 0), with current value equal to 2.5 (fourth constructor argument set to 2.5). This specific constructor defines identity functions, and identity functions are the trick we use to represent variables (there are of course other constructors, for example to build constants or functions from all their derivatives if they are known beforehand). From the user point of view, the xinstance can be seen as the x variable, but it is really the identity function applied to free parameter number 0. As the identity function, it has the same value as its parameter, its first derivative is 1.0 with respect to this free parameter, and all its higher order derivatives are 0.0. This can be checked by calling the getValue() or getPartialDerivative() methods on x.

When we compute y from this setting, what we really do is chain f after the identity function, so the net result is that the derivatives are computed with respect to the indexed free parameters (i.e. only free parameter number 0 here since there is only one free parameter) of the identity function x. Going one step further, if we compute z = g(y), we will also compute z as a function of the initial free parameter. The very important consequence is that if we call z.getPartialDerivative(1), we will not get the first derivative ofg with respect to y, but with respect to the free parameter p0: the derivatives of g and f *will* be chained together automatically, without user intervention.

This design choice is a very classical one in many algorithmic differentiation frameworks, either based on operator overloading (like the one we implemented here) or based on code generation. It implies the user has to *bootstrap* the system by providing initial derivatives, and this is essentially done by setting up identity function, i.e. functions that represent the variables themselves and have only unit first derivative.

This design also allow a very interesting feature which can be explained with the following example. Suppose we have a two arguments function f and a one argument function g. If we compute g(f(x, y)) with x andy be two variables, we want to be able to compute the partial derivatives dg/dx, dg/dy, d2g/dx2 d2g/dxdyd2g/dy2. This does make sense since we combined the two functions, and it does make sense despite g is a one argument function only. In order to do this, we simply set up x as an identity function of an implicit free parameter p0 and y as an identity function of a different implicit free parameter p1 and compute everything directly. In order to be able to combine everything, however, both x and y must be built with the appropriate dimensions, so they will both be declared to handle two free parameters, but x will depend only on parameter 0 while y will depend on parameter 1. Here is how we do this (note that getPartialDerivative is a variable arguments method which take as arguments the derivation order with respect to all free parameters, i.e. the first argument is derivation order with respect to free parameter 0 and the second argument is derivation order with respect to free parameter 1):

```
int params = 2;
int order = 2;
double xRealValue =  2.5;
double yRealValue = -1.3;
DerivativeStructure x = new DerivativeStructure(params, order, 0, xRealValue);
DerivativeStructure y = new DerivativeStructure(params, order, 1, yRealValue);
DerivativeStructure f = DerivativeStructure.hypot(x, y);
DerivativeStructure g = f.log();
System.out.println("g        = " + g.getValue();
System.out.println("dg/dx    = " + g.getPartialDerivative(1, 0);
System.out.println("dg/dy    = " + g.getPartialDerivative(0, 1);
System.out.println("d2g/dx2  = " + g.getPartialDerivative(2, 0);
System.out.println("d2g/dxdy = " + g.getPartialDerivative(1, 1);
System.out.println("d2g/dy2  = " + g.getPartialDerivative(0, 2);
```

There are several ways a user can create an implementation of the UnivariateDifferentiableFunctioninterface. The first method is to simply write it directly using the appropriate methods fromDerivativeStructure to compute addition, subtraction, sine, cosine... This is often quite straigthforward and there is no need to remember the rules for differentiation: the user code only represent the function itself, the differentials will be computed automatically under the hood. The second method is to write a classicalUnivariateFunction and to pass it to an existing implementation of the UnivariateFunctionDifferentiatorinterface to retrieve a differentiated version of the same function. The first method is more suited to small functions for which user already control all the underlying code. The second method is more suited to either large functions that would be cumbersome to write using the DerivativeStructure API, or functions for which user does not have control to the full underlying code (for example functions that call external libraries).

Apache Commons Math provides one implementation of the UnivariateFunctionDifferentiator interface:FiniteDifferencesDifferentiator. This class creates a wrapper that will call the user-provided function on a grid sample and will use finite differences to compute the derivatives. It takes care of boundaries if the variable is not defined on the whole real line. It is possible to use more points than strictly required by the derivation order (for example one can specify an 8-points scheme to compute first derivative only). However, one must be aware that tuning the parameters for finite differences is highly

problem-dependent. Choosing the wrong step size or the wrong number of sampling points can lead to huge errors. Finite differences are also not well suited to compute high order derivatives.

Another implementation of the UnivariateFunctionDifferentiator interface is under development in the related project Apache Commons Nabla. This implementation uses automatic code analysis and generation at binary level. However, at time of writing (end 2012), this project is not yet suitable for production use.

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.

Sections 2.2-2.6 below 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 in 2.7. 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 nextXxxmethods 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 JDKSecureRandom PRNG to generate cryptographically secure sequences. ThesetSecureAlgorithm 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. The ISAACRandom class implements a fast cryptographically secure pseudorandom numbers generator.

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:

```
RandomData randomData = new RandomDataImpl();
for (int i = 0; i < 1000; i++) {
   value = randomData.nextLong(1, 1000000);
}
```

The following will not in general produce a good random sequence, since the PRNG is reseeded each time through the loop with the current time in milliseconds:

```
for (int i = 0; i < 1000; i++) {
   RandomDataImpl randomData = new RandomDataImpl();
   value = randomData.nextLong(1, 1000000);
}
```

The following will produce the same random sequence each time it is executed:

```
RandomData randomData = new RandomDataImpl();
randomData.reSeed(1000);
for (int i = 0; i = 1000; i++) {
   value = randomData.nextLong(1, 1000000);
}
```

The following will produce a different random sequence each time it is executed.

```
RandomData randomData = new RandomDataImpl();
randomData.reSeedSecure(1000);
for (int i = 0; i < 1000; i++) {
   value = randomData.nextSecureLong(1, 1000000);
}
```

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.

Generating random vectors from a bivariate normal distribution

```
// Create and seed a RandomGenerator (could use any of the generators in the random package here)
RandomGenerator rg = new JDKRandomGenerator();
rg.setSeed(17399225432l);  // Fixed seed means same results every time
// Create a GassianRandomGenerator using rg as its source of randomness
GaussianRandomGenerator rawGenerator = new GaussianRandomGenerator(rg);
// Create a CorrelatedRandomVectorGenerator using rawGenerator for the components
CorrelatedRandomVectorGenerator generator =
     new CorrelatedRandomVectorGenerator(mean, covariance, 1.0e-12 * covariance.getNorm(), rawGenerator);
// Use the generator to generate correlated vectors
double[] randomVector = generator.nextVector();
...
```

The mean argument is a double[] array 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

```
double[] mean = {1, 2};
double[][] cov = {{9, c}, {c, 16}};
RealMatrix covariance = MatrixUtils.createRealMatrix(cov);
```

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 NormalizedRandomGeneratormay 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 ofnextHexString(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, nextSecureHexStringgenerates 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 nextSamplemethod in the RandomData interface. Specifically, if c is a collection containing at least kobjects, 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, thenrandomData.nextPermutation(n, k) returns an int[] array of length k whose 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 will read data from 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.

```
ValueServer vs = new ValueServer();
vs.setValuesFileURL(url);
vs.setMode(ValueServer.REPLAY_MODE);
vs.resetReplayFile();
double 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:

```
ValueServer vs = new ValueServer();
vs.setValuesFileURL(url);
vs.setMode(ValueServer.DIGEST_MODE);
vs.computeDistribution(500); //Read file and estimate distribution using 500 bins
double value = vs.getNext();
// ...Generate and use more values...
```

See the javadoc for ValueServer and EmpiricalDistribution for more details. 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. TheRandomGenerator 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, theRandomAdaptor 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 RandomGeneratorinterface:

- 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:
    - MersenneTwister
    - 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). The errata for the paper are in wellrng-errata.txt.

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 exemple 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 then MersenneTwister (the paper describing than 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 Well44497bare 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 ...). 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.
 */
public class RngPackGenerator extends AbstractRandomGenerator {

   private RanMT random = new RanMT();

   public void setSeed(long seed) {
     random = new RanMT(seed);
   }

   public double nextDouble() {
      return random.raw();
   }

   public double nextGaussian() {
      return random.gaussian();
   }

   public int nextInt(int n) {
      return random.choose(n);
   }

   public boolean nextBoolean() {
      return random.coin();
   }
}
```

Use the Mersenne Twister RandomGenerator in place of java.util.Random inRandomData

```
RandomData randomData = new RandomDataImpl(new RngPackGenerator());
```

Create an adaptor instance based on the Mersenne Twister generator that can be used in place of a Random

```
RandomGenerator generator = new RngPackGenerator();
Random 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
```

Special Functions

5.1 Overview

The special package of Commons-Math gathers several useful special functions not provided by java.lang.Math.

5.2 Erf functions

Erf contains several useful functions involving the Error Function, Erf.

Function	Method	Reference
Error Function	erf	See Erf from MathWorld

5.3 Gamma functions

Class Gamma contains several useful functions involving the Gamma Function.

Gamma

Gamma.gamma(x) computes the Gamma function, Γ(x) (see MathWorld, DLMF). The accuracy of the Commons-Math implementation is assessed by comparaison with high precision values computed with the Maxima Computer Algebra System.

Interval	Values tested	Average error	Standard deviation	Maximum error
-5 < x < -4	x[i] = i / 1024, i = -5119, ..., -4097	0.49 ulps	0.57 ulps	3.0 ulps
-4 < x < -3	x[i] = i / 1024, i = -4095, ..., -3073	0.36 ulps	0.51 ulps	2.0 ulps
-3 < x < -2	x[i] = i / 1024, i = -3071, ..., -2049	0.41 ulps	0.53 ulps	2.0 ulps
-2 < x < -1	x[i] = i / 1024, i = -2047, ..., -1025	0.37 ulps	0.50 ulps	2.0 ulps
-1 < x < 0	x[i] = i / 1024, i = -1023, ..., -1	0.46 ulps	0.54 ulps	2.0 ulps
0 < x ≤ 8	x[i] = i / 1024, i = 1, ..., 8192	0.33 ulps	0.48 ulps	2.0 ulps
8 < x ≤ 141	x[i] = i / 64, i = 513, ..., 9024	1.32 ulps	1.19 ulps	7.0 ulps

Log Gamma

Gamma.logGamma(x) computes the natural logarithm of the Gamma function, log Γ(x), for x > 0 (see MathWorld, DLMF). The accuracy of the Commons-Math

implementation is assessed by comparaison with high precision values computed with the Maxima Computer Algebra System.

Interval	Values tested	Average error	Standard deviation	Maximum error
0 < x ≤ 8	x[i] = i / 1024, i = 1, ..., 8192	0.32 ulps	0.50 ulps	4.0 ulps
8 < x ≤ 1024	x[i] = i / 8, i = 65, ..., 8192	0.43 ulps	0.53 ulps	3.0 ulps
1024 < x ≤ 8192	x[i], i = 1025, ..., 8192	0.53 ulps	0.56 ulps	3.0 ulps
8933.439345993791 ≤ x ≤ 1.75555970201398e+305	x[i] = 2**(i / 8), i = 105, ..., 8112	0.35 ulps	0.49 ulps	2.0 ulps

Regularized Gamma

Gamma.regularizedGammaP(a, x) computes the value of the regularized Gamma function, P(a, x) (see MathWorld).

5.4 Beta functions

Beta contains several useful functions involving the Beta Function.

Log Beta

Beta.logBeta(a, b) computes the value of the natural logarithm of the Beta function, log B(a, b). (see MathWorld, DLMF). The accuracy of the Commons-Math implementation is assessed by comparaison with high precision values computed with the Maxima Computer Algebra System.

Interval	Values tested	Average error	Standard deviation	Maximum error
0 < x ≤ 8 0 < y ≤ 8	x[i] = i / 32, i = 1, ..., 256 y[j] = j / 32, j = 1, ..., 256	1.80 ulps	81.08 ulps	14031.0 ulps
0 < x ≤ 8 8 < y ≤ 16	x[i] = i / 32, i = 1, ..., 256 y[j] = j / 32, j =	0.50 ulps	3.64 ulps	694.0 ulps

	257, ..., 512			
0 < x ≤ 8 16 < y ≤ 256	x[i] = i / 32, i = 1, ..., 256 y[j] = j, j = 17, ..., 256	1.04 ulps	139.32 ulps	34509.0 ulps
8 < x ≤ 16 8 < y ≤ 16	x[i] = i / 32, i = 257, ..., 512 y[j] = j / 32, j = 257, ..., 512	0.35 ulps	0.48 ulps	2.0 ulps
8 < x ≤ 16 16 < y ≤ 256	x[i] = i / 32, i = 257, ..., 512 y[j] = j, j = 17, ..., 256	0.31 ulps	0.47 ulps	2.0 ulps
16 < x ≤ 256 16 < y ≤ 256	x[i] = i, i = 17, ..., 256 y[j] = j, j = 17, ..., 256	0.35 ulps	0.49 ulps	2.0 ulps

Regularized Beta

(see MathWorld)

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 ...) 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:

```
import org.apache.commons.math3.linear.*

// Create a real matrix with two rows and three columns, using a factory
// method that selects the implementation class for us.
matrixData = [[1d,2d,3d], [2d,5d,3d]] as double [][]

mm = MatrixUtils.createRealMatrix(matrixData)

// One more with three rows, two columns, this time instantiating the
// RealMatrix implementation class directly.
matrixData2 = [[1d,2d], [2d,5d], [1d, 7d]] as double[][]
nn = new Array2DRowRealMatrix(matrixData2)
// Note: The constructor copies  the input double[][] array in both cases.
// Now multiply m by n
pp = mm.multiply(nn)
println(pp.getRowDimension())    // 2
println(pp.getColumnDimension())  // 2
// Invert p, using LU decomposition
pInverse = new LUDecomposition(pp).getSolver().getInverse()
```

The three main implementations of the interface are Array2DRowRealMatrix andBlockRealMatrix 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 ...)
- 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 linear sense or in least square sense. ARealMatrix 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

$$
\begin{aligned}
2x + 3y - 2z &= 1 \\
-x + 7y + 6x &= -2 \\
4x - 3y - 5z &= 1
\end{aligned}
$$

Start by decomposing the coefficient matrix A (in this case using LU decomposition) and build a solver

```
coefficients =     new Array2DRowRealMatrix([[ 2, 3, -2 ],  [-1, 7, 6 ], [ 4, -3, -5 ]] as double [][], false)
solver = new LUDecomposition(coefficients).getSolver()
```

Next create a RealVector array to represent the constant vector B and usesolve(RealVector) to solve the system

```
constants = new ArrayRealVector( [1, -2, 1 ] as double [], false)
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 algorithms		
Name	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 solution

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 for the systems 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 $A \cdot X = \lambda \cdot X$, where λ is a real scalar.

The *getSingularValues()*, *getU()*, *getS()* and *getV()* methods of the *SingularValueDecomposition* interface allow to solve singular values problems of the form $A \cdot X_i = \lambda \cdot Y_i$ where λ is a real scalar, and where the X_i and Y_i vectors form orthogonal bases of their respective vector spaces (which may have different dimensions).

Non-real fields (complex, fractions ...)

In addition to the real field, matrices and vectors using non-real field elements can be used. The fields already supported by the library are:

- Complex
- Fraction
- BigFraction
- BigReal

Utilities

The org.apache.commons.math3.util package collects a group of array utilities, value transformers, and numerical routines used by implementation classes in commons-math.

Double array utilities

To maintain statistics based on a "rolling" window of values, a resizable array implementation was developed and is provided for reuse in the util package. The core functionality provided is described in the documentation for the interface, DoubleArray. This interface adds one method, addElementRolling(double) to basic list accessors. The addElementRolling method adds an element (the actual parameter) to the end of the list and removes the first element in the list.

The ResizableDoubleArray class provides a configurable, array-backed implementation of the DoubleArray interface. When addElementRolling is invoked, the underlying array is expanded if necessary, the new element is added to the end of the array and the "usable window" of the array is moved forward, so that the first element is effectively discarded, what was the second becomes the first, and so on. To efficiently manage storage, two maintenance operations need to be periodically performed -- orphaned elements at the beginning of the array need to be reclaimed and space for new elements at the end needs to be created. Both of these operations are handled automatically, with frequency / effect driven by the configuration properties expansionMode, expansionFactor andcontractionCriteria. See ResizableDoubleArray for details.

int/double hash map

The OpenIntToDoubleHashMap class provides a specialized hash map implementation for int/double. This implementation has a much smaller memory overhead than standardjava.util.HashMap class. It uses open addressing and primitive arrays, which greatly reduces the number of intermediate objects and improve data locality.

Continued Fractions

The ContinuedFraction class provides a generic way to create and evaluate continued fractions. The easiest way to create a continued fraction is to subclassContinuedFraction and override the getA and getB methods which return the continued fraction terms. The precise definition of these terms is explained in Continued Fraction, equation (1) from MathWorld.

As an example, the constant Pi can be computed using a continued fraction. The following anonymous class provides the implementation:

```
ContinuedFraction c = new ContinuedFraction() {
  public double getA(int n, double x) {
    switch(n) {
      case 0: return 3.0;
      default: return 6.0;
    }
  }

  public double getB(int n, double x) {
    double y = (2.0 * n) - 1.0;
    return y * y;
  }
}
```

Then, to evalute Pi, simply call any of the evalute methods (Note, the point of evalution in this example is meaningless since Pi is a constant).

For a more practical use of continued fractions, consider the exponential function. The following anonymous class provides its implementation:

```
ContinuedFraction c = new ContinuedFraction() {
    public double getA(int n, double x) {
        if (n % 2 == 0) {
            switch(n) {
                case 0: return 1.0;
                default: return 2.0;
            }
        } else {
            return n;
        }
    }

    public double getB(int n, double x) {
        if (n % 2 == 0) {
            return -x;
        } else {
            return x;
        }
    }
}
```

Then, to evalute e^x for any value x, simply call any of the evalute methods.

Binomial coefficients, factorials, Stirling numbers and other common math functions

A collection of reusable math functions is provided in the ArithmeticUtils utility class. ArithmeticUtils currently includes methods to compute the following:

- Binomial coefficients -- "n choose k" available as an (exact) long value,binomialCoefficient(int, int) for small n, k; as a double,binomialCoefficientDouble(int, int) for larger values; and in a "super-sized" version, binomialCoefficientLog(int, int) that returns the natural logarithm of the value.
- Stirling numbers of the second kind -- S(n,k) as an exact long valuestirlingS2(int, int) for small n, k.
- Factorials -- like binomial coefficients, these are available as exact long values,factorial(int); doubles, factorialDouble(int); or logs, factorialLog(int).
- Least common multiple and greatest common denominator functions.

Fast mathematical functions

Apache Commons Math provides a faster, more accurate, portable alternative to the regular Math and StrictMathclasses for large scale computation.

FastMath is a drop-in replacement for both Math and StrictMath. This means that for any method in Math (say Math.sin(x) or Math.cbrt(y)), user can directly change the class and use the methods as is (using FastMath.sin(x) or FastMath.cbrt(y) in the previous example).

FastMath speed is achieved by relying heavily on optimizing compilers to native code present in many JVM todays and use of large tables. Precomputed literal arrays are provided in this class to speed up load time. These precomputed tables are used in the default configuration, to improve speed even at first use of the class. If users prefer to compute the tables automatically at load time, they can change a compile-time

constant. This will increase class load time at first use, but this overhead will occur only once per run, regardless of the number of subsequent calls to computation methods. Note that FastMath is extensively used inside Apache Commons Math, so by calling some algorithms, the one-shot overhead when the constant is set to false will occur regardless of the end-user calling FastMath methods directly or not. Performance figures for a specific JVM and hardware can be evaluated by running the FastMathTestPerformance tests in the test directory of the source distribution.

FastMath accuracy should be mostly independent of the JVM as it relies only on IEEE-754 basic operations and on embedded tables. Almost all operations are accurate to about 0.5 ulp throughout the domain range. This statement, of course is only a rough global observed behavior, it is *not* a guarantee for *every* double numbers input (see William Kahan's Table Maker's Dilemma).

FastMath additionally implements the following methods not found in Math/StrictMath:

- asinh(double)
- acosh(double)
- atanh(double)
- pow(double,int)

The following methods are found in Math/StrictMath since 1.6 only, they are provided by FastMath even in 1.5 Java virtual machines

- copySign(double, double)
- getExponent(double)
- nextAfter(double,double)
- nextUp(double)
- scalb(double, int)
- copySign(float, float)
- getExponent(float)
- nextAfter(float,double)
- nextUp(float)
- scalb(float, int)

Miscellaneous

The MultidimensionalCounter is a utility class that converts a set of indices (identifying points in a multidimensional space) to a single index (e.g. identifying a location in a one-dimensional array).

Complex Numbers

Overview

The complex packages provides a complex number type as well as complex versions of common transcendental functions and complex number formatting.

Complex Numbers

Complex provides a complex number type that forms the basis for the complex functionality found in commons-math.

Complex functions and arithmetic operations are implemented in commons-math by applying standard computational formulas and following the rules for java.lang.Double arithmetic in handling infinite and NaN values. No attempt is made to comply with ANSII/IEC C99x Annex G or any other standard for Complex arithmetic. See the class and method javadocs for the Complex and ComplexUtils classes for details on computing formulas.

To create a complex number, simply call the constructor passing in two floating-point arguments, the first being the real part of the complex number and the second being the imaginary part:

```
import org.apache.commons.math3.complex.*

c = new Complex(1.0, 3.0)
```

Complex numbers may also be created from polar representations using the *polar2Complex* method in ComplexUtils.

The Complex class provides basic unary and binary complex number operations. These operations provide the means to add, subtract, multiply and divide complex numbers along with other complex number functions similar to the real number functions found injava.math.BigDecimal:

```
lhs = new Complex(1.0, 3.0)

rhs = new Complex(2.0, 5.0)

answer = lhs.add(rhs)       // add two complex numbers

answer = lhs.subtract(rhs)  // subtract two complex numbers

answer = lhs.abs()          // absolute value

answer = lhs.conjugate(rhs) // complex conjugate
```

Complex Transcendental Functions

Complex also provides implementations of several transcendental functions involving complex number arguments. Prior to version 1.2, these functions were provided by ComplexUtils in a way similar to the real number functions found in java.lang.Math, but this has been deprecated. These operations provide the means to compute the log, sine, tangent, and other complex values :

first = new Complex(1.0, 3.0)

second = new Complex(2.0, 5.0)

answer = first.log() // natural logarithm.

answer = first.cos() // cosine

answer = first.pow(second) // first raised to the power of second

Complex Formatting and Parsing

Complex instances can be converted to and from strings using the ComplexFormat class.ComplexFormat is a java.text.Format extension and, as such, is used like other formatting objects (e.g. java.text.SimpleDateFormat):

```
ComplexFormat format = new ComplexFormat(); // default format
Complex c = new Complex(1.1111, 2.2222);
String s = format.format(c); // s contains "1.11 + 2.22i"
```

To customize the formatting output, one or two java.text.NumberFormat instances can be used to construct a ComplexFormat. These number formats control the formatting of the real and imaginary values of the complex number:

```
NumberFormat nf = NumberFormat.getInstance();
nf.setMinimumFractionDigits(3);
nf.setMaximumFractionDigits(3);
// create complex format with custom number format
// when one number format is used, both real and
// imaginary parts are formatted the same
ComplexFormat cf = new ComplexFormat(nf);
Complex c = new Complex(1.11, 2.2222);
String s = format.format(c); // s contains "1.110 + 2.222i"
NumberFormat nf2 = NumberFormat.getInstance();
nf.setMinimumFractionDigits(1);
nf.setMaximumFractionDigits(1);
// create complex format with custom number formats
cf = new ComplexFormat(nf, nf2);
s = format.format(c); // s contains "1.110 + 2.2i"
```

Another formatting customization provided by ComplexFormat is the text used for the imaginary designation. By default, the imaginary notation is "i" but, it can be manipulated using the setImaginaryCharacter method.

Formatting inverse operation, parsing, can also be performed by ComplexFormat. Parse a complex number from a string, simply call the parse method:

```
ComplexFormat cf = new ComplexFormat();
Complex c = cf.parse("1.110 + 2.222i");
```

Probability Distributions

8.1 Overview

The distributions package provide a framework for some commonly used probability distributions.

8.2 Distribution Framework

The distribution framework provides the means to compute probability density function (PDF) probabilities and cumulative distribution function (CDF) probabilities for common probability distributions. Along with the direct computation of PDF and CDF probabilities, the framework also allows for the computation of inverse PDF and inverse CDF values.

Using a distribution object, PDF and CDF probabilities are easily computed using thecumulativeProbability methods. For a distribution X, and a domain value, x,cumulativeProbability computes P(X <= x) (i.e. the lower tail probability of X).

```
TDistribution t = new TDistribution(29);
double lowerTail = t.cumulativeProbability(-2.656);
// P(T <= -2.656)
double upperTail = 1.0 -
t.cumulativeProbability(2.75); // P(T >= 2.75)
```

The inverse PDF and CDF values are just as easily computed using theinverseCumulativeProbability methods. For a distribution X, and a probability, p,inverseCumulativeProbability computes the domain value x, such that:

- P(X <= x) = p, for continuous distributions
- P(X <= x) <= p, for discrete distributions

Notice the different cases for continuous and discrete distributions. This is the result of PDFs not being invertible functions. As such, for discrete distributions, an exact domain value can not be returned. Only the "best" domain value. For Commons-Math, the "best" domain value is determined by the largest domain value whose cumulative probability is less-than or equal to the given probability.

8.3 User Defined Distributions

Since there are numerous distributions and Commons-Math only directly supports a handful, it may be necessary to extend the distribution framework to satisfy individual needs. It is recommended that the Distribution, ContinuousDistribution,DiscreteDistribution, and IntegerDistribution interfaces serve as base types for any extension. These serve as the basis for all the distributions directly supported by Commons-Math and using those interfaces for implementation purposes will ensure any extension is compatible with the remainder of Commons-Math. To aid in implementing a distribution extension, the AbstractDistribution, AbstractContinuousDistribution,

andAbstractIntegerDistribution provide implementation building blocks and offer basic distribution functionality. By extending these abstract classes directly, much of the repetitive distribution implementation is already developed and should save time and effort in developing user-defined distributions.

Fractions

The fraction packages provides a fraction number type as well as fraction number formatting.

9.2 Fraction Numbers

Fraction and BigFraction provide fraction number type that forms the basis for the fraction functionality found in Commons-Math. The former one can be used for fractions whose numerators and denominators are small enough to fit in an int (taking care of intermediate values) while the second class should be used when there is a risk the numerator and denominator grow very large.

A fraction number, can be built from two integer arguments representing numerator and denominator or from a double which will be approximated:

```
Fraction f = new Fraction(1, 3); // 1 / 3
Fraction g = new Fraction(0.25); // 1 / 4
```

Of special note with fraction construction, when a fraction is created it is always reduced to lowest terms.

The Fraction class provides many unary and binary fraction operations. These operations provide the means to add, subtract, multiple and, divide fractions along with other functions similar to the real number functions found in java.math.BigDecimal:

```
Fraction lhs = new Fraction(1, 3);
Fraction rhs = new Fraction(2, 5);
Fraction answer = lhs.add(rhs);     // add two fractions
        answer = lhs.subtract(rhs); // subtract two fractions
        answer = lhs.abs();         // absolute value
        answer = lhs.reciprocal();  // reciprocal of lhs
```

Like fraction construction, for each of the fraction functions, the resulting fraction is reduced to lowest terms.

9.3 Fraction Formatting and Parsing

Fraction instances can be converted to and from strings using the FractionFormatclass. FractionFormat is a java.text.Format extension and, as such, is used like other formatting objects (e.g. java.text.SimpleDateFormat):

```
FractionFormat format = new FractionFormat(); // default format
Fraction f = new Fraction(2, 4);
String s = format.format(f); // s contains "1 / 2", note the reduced
fraction
```

To customize the formatting output, one or two java.text.NumberFormat instances can be used to construct a FractionFormat. These number formats control the formatting of the numerator and denominator of the fraction:

```
NumberFormat nf = NumberFormat.getInstance(Locale.FRANCE);
// create fraction format with custom number format
// when one number format is used, both numerator and
// denominator are formatted the same
FractionFormat format = new FractionFormat(nf);
```

```
Fraction f = new Fraction(2000, 3333);
String s = format.format(c); // s contains "2.000 / 3.333"
NumberFormat nf2 = NumberFormat.getInstance(Locale.US);
// create fraction format with custom number formats
format = new FractionFormat(nf, nf2);
s = format.format(f); // s contains "2.000 / 3,333"
```

Formatting's inverse operation, parsing, can also be performed by FractionFormat. To parse a fraction from a string, simply call the parse method:

```
FractionFormat ff = new FractionFormat();
Fraction f = ff.parse("-10 / 21");
```

Transforms

org.apache.commons.math3.transform
Class FastFourierTransformer

java.lang.Object
  org.apache.commons.math3.transform.FastFourierTransformer

All Implemented Interfaces:
    Serializable

---

public class `FastFourierTransformer` extends `Object`implements `Serializable`

Implements the Fast Fourier Transform for transformation of one-dimensional real or complex data sets. For reference, seeApplied Numerical Linear Algebra, ISBN 0898713897, chapter 6.

There are several variants of the discrete Fourier transform, with various normalization conventions, which are specified by the parameter DftNormalization.

The current implementation of the discrete Fourier transform as a fast Fourier transform requires the length of the data set to be a power of 2. This greatly simplifies and speeds up the code. Users can pad the data with zeros to meet this requirement. There are other flavors of FFT, for reference, see S. Winograd, *On computing the discrete Fourier transform*, Mathematics of Computation, 32 (1978), 175 - 199.

Since:
    1.2
Version:
    $Id: FastFourierTransformer.java 1385310 2012-09-16 16:32:10Z tn $
See Also:
    DftNormalization, Serialized Form

Constructor Summary
FastFourierTransformer(DftNormalization normalization) Creates a new instance of this class, with various normalization conventions.

Method Summary	
Object	**mdfft**(Object mdca, TransformType type) **Deprecated.** *see MATH-736*
Complex[]	**transform**(Complex[] f, TransformType type) Returns the (forward, inverse) transform of the specified complex data set.
Complex[]	**transform**(double[] f, TransformType type) Returns the (forward, inverse) transform of the specified real data set.
Complex[]	**transform**(UnivariateFunction f, double min, double max, int n, TransformType type) Returns the (forward, inverse) transform of the specified real function, sampled on the specified interval.

static void	**transformInPlace**(double[][] dataRI, DftNormalization normalization,TransformType type) Computes the standard transform of the specified complex data.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

FastFourierTransformer

public **FastFourierTransformer**(DftNormalization normalization)

Creates a new instance of this class, with various normalization conventions.

Parameters:
normalization - the type of normalization to be applied to the transformed data

Method Detail

transformInPlace

```
public static void transformInPlace(double[][] dataRI,
                                    DftNormalization normalization,
                                    TransformType type)
```

Computes the standard transform of the specified complex data. The computation is done in place. The input data is laid out as follows

- dataRI[0][i] is the real part of the i-th data point,
- dataRI[1][i] is the imaginary part of the i-th data point.

Parameters:
dataRI - the two dimensional array of real and imaginary parts of the data
normalization - the normalization to be applied to the transformed data
type - the type of transform (forward, inverse) to be performed

Throws:
DimensionMismatchException - if the number of rows of the specified array is not two, or the array is not rectangular
MathIllegalArgumentException - if the number of data points is not a power of two

---

transform

```
public Complex[] transform(double[] f,
                           TransformType type)
```

Returns the (forward, inverse) transform of the specified real data set.

Parameters:
f - the real data array to be transformed
type - the type of transform (forward, inverse) to be performed

Returns:
the complex transformed array

Throws:

MathIllegalArgumentException - if the length of the data array is not a power of two

---

transform

```
public Complex[] transform(UnivariateFunction f,
                           double min,
                           double max,
                           int n,
                           TransformType type)
```

Returns the (forward, inverse) transform of the specified real function, sampled on the specified interval.

Parameters:
f - the function to be sampled and transformed
min - the (inclusive) lower bound for the interval
max - the (exclusive) upper bound for the interval
n - the number of sample points
type - the type of transform (forward, inverse) to be performed

Returns:
the complex transformed array

Throws:
NumberIsTooLargeException - if the lower bound is greater than, or equal to the upper bound
NotStrictlyPositiveException - if the number of sample points n is negative
MathIllegalArgumentException - if the number of sample points n is not a power of two

---

transform

```
public Complex[] transform(Complex[] f,
                           TransformType type)
```

Returns the (forward, inverse) transform of the specified complex data set.

Parameters:
f - the complex data array to be transformed
type - the type of transform (forward, inverse) to be performed

Returns:
the complex transformed array

Throws:
MathIllegalArgumentException - if the length of the data array is not a power of two

---

mdfft

```
@Deprecated
public Object mdfft(Object mdca,
                    TransformType type)
```

Deprecated. *see MATH-736*

Performs a multi-dimensional Fourier transform on a given array. Use transform(Complex[], TransformType) in a row-column implementation in any number of dimensions with O(N×log(N))

complexity with $N = n_1 \times n_2 \times n_3 \times ... \times n_d$, where n_k is the number of elements in dimension k, and d is the total number of dimensions.

Parameters:

mdca - Multi-Dimensional Complex Array, i.e. Complex[][][][]

type - the type of transform (forward, inverse) to be performed

Returns:

transform of mdca as a Multi-Dimensional Complex Array, i.e. Complex[][][][]

Throws:

IllegalArgumentException - if any dimension is not a power of two

org.apache.commons.math3.transform
Class FastCosineTransformer

java.lang.Object
org.apache.commons.math3.transform.FastCosineTransformer

All Implemented Interfaces:

Serializable, RealTransformer

---

public class **FastCosineTransformer**extends `Object`implements `RealTransformer`, `Serializable`

Implements the Fast Cosine Transform for transformation of one-dimensional real data sets. For reference, see James S. Walker, Fast Fourier Transforms, chapter 3 (ISBN 0849371635).

There are several variants of the discrete cosine transform. The present implementation corresponds to DCT-I, with various normalization conventions, which are specified by the parameter DctNormalization.

DCT-I is equivalent to DFT of an even extension of the data series. More precisely, if $x_0, \ldots, x_{N-1}$ is the data set to be cosine transformed, the extended data set $x_0^\#, \ldots, x_{2N-3}^\#$ is defined as follows

- $x_k^\# = x_k$ if $0 \le k < N$,
- $x_k^\# = x_{2N-2-k}$ if $N \le k < 2N - 2$.

Then, the standard DCT-I $y_0, \ldots, y_{N-1}$ of the real data set $x_0, \ldots, x_{N-1}$ is equal to half of the N first elements of the DFT of the extended data set $x_0^\#, \ldots, x_{2N-3}^\#$

$y_n = (1 / 2) \sum_{k=0}^{2N-3} x_k^\# \exp[-2\pi i\, nk / (2N - 2)] \quad k = 0, \ldots, N-1.$

The present implementation of the discrete cosine transform as a fast cosine transform requires the length of the data set to be a power of two plus one ($N = 2^n + 1$). Besides, it implicitly assumes that the sampled function is even.

Since:

1.2

Version:

$Id: FastCosineTransformer.java 1385310 2012-09-16 16:32:10Z tn $

See Also:

Serialized Form

Constructor Summary
FastCosineTransformer(DctNormalization normalization) Creates a new instance of this class, with various normalization conventions.

Method Summary	
protected double[]	**fct**(double[] f) Perform the FCT algorithm (including inverse).
double[]	**transform**(double[] f, TransformType type) Returns the (forward, inverse) transform of the specified real data set.
double[]	**transform**(UnivariateFunction f, double min, double max, int n, TransformType type) Returns the (forward, inverse) transform of the specified real function, sampled on the specified interval.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

FastCosineTransformer

public **FastCosineTransformer**(`DctNormalization` normalization)

Creates a new instance of this class, with various normalization conventions.

Parameters:
normalization - the type of normalization to be applied to the transformed data

Method Detail

transform

public double[] **transform**(double[] f,
`TransformType` type)
throws `MathIllegalArgumentException`

Returns the (forward, inverse) transform of the specified real data set.

Specified by:
transform in interface RealTransformer

Parameters:
f - the real data array to be transformed (signal)
type - the type of transform (forward, inverse) to be performed

Returns:
the real transformed array (spectrum)

Throws:
MathIllegalArgumentException - if the length of the data array is not a power of two plus one

transform

```
public double[] transform(UnivariateFunction f,
                          double min,
                          double max,
                          int n,
                          TransformType type)
                   throws MathIllegalArgumentException
```

Returns the (forward, inverse) transform of the specified real function, sampled on the specified interval.

Specified by:
transform in interface RealTransformer

Parameters:
f - the function to be sampled and transformed
min - the (inclusive) lower bound for the interval
max - the (exclusive) upper bound for the interval
n - the number of sample points
type - the type of transform (forward, inverse) to be performed

Returns:
the real transformed array

Throws:
NonMonotonicSequenceException - if the lower bound is greater than, or equal to the upper bound
NotStrictlyPositiveException - if the number of sample points is negative
MathIllegalArgumentException - if the number of sample points is not a power of two plus one

---

fct

```
protected double[] fct(double[] f)
                throws MathIllegalArgumentException
```

Perform the FCT algorithm (including inverse).

Parameters:
f - the real data array to be transformed

Returns:
the real transformed array

Throws:
MathIllegalArgumentException - if the length of the data array is not a power of two plus one

org.apache.commons.math3.transform
Class FastSineTransformer

java.lang.Object
  org.apache.commons.math3.transform.FastSineTransformer

All Implemented Interfaces:
 Serializable, RealTransformer

---

public class **FastSineTransformer**extends `Object`implements `RealTransformer`, `Serializable`

Implements the Fast Sine Transform for transformation of one-dimensional real data sets. For reference, see James S. Walker,Fast Fourier Transforms, chapter 3 (ISBN 0849371635).

There are several variants of the discrete sine transform. The present implementation corresponds to DST-I, with various normalization conventions, which are specified by the parameter DstNormalization. It should be noted that regardless to the convention, the first element of the dataset to be transformed must be zero.

DST-I is equivalent to DFT of an odd extension of the data series. More precisely, if $x_0, \ldots, x_{N-1}$ is the data set to be sine transformed, the extended data set $x_0^\#, \ldots, x_{2N-1}^\#$ is defined as follows

- $x_0^\# = x_0 = 0$,
- $x_k^\# = x_k$ if $1 \leq k < N$,
- $x_N^\# = 0$,
- $x_k^\# = -x_{2N-k}$ if $N + 1 \leq k < 2N$.

Then, the standard DST-I $y_0, \ldots, y_{N-1}$ of the real data set $x_0, \ldots, x_{N-1}$ is equal to half of i (the pure imaginary number) times the N first elements of the DFT of the extended data set $x_0^\#, \ldots, x_{2N-1}^\#$
$y_n = (i / 2) \sum_{k=0}^{2N-1} x_k^\# \exp[-2\pi i\, nk / (2N)] \quad k = 0, \ldots, N-1.$

The present implementation of the discrete sine transform as a fast sine transform requires the length of the data to be a power of two. Besides, it implicitly assumes that the sampled function is odd. In particular, the first element of the data set must be 0, which is enforced in transform(UnivariateFunction, double, double, int, TransformType), after sampling.

Since:
 1.2
Version:
 $Id: FastSineTransformer.java 1385310 2012-09-16 16:32:10Z tn $

See Also:
Serialized Form

Constructor Summary
FastSineTransformer(DstNormalization normalization) Creates a new instance of this class, with various normalization conventions.

Method Summary	
protected double[]	**fst**(double[] f) Perform the FST algorithm (including inverse).
double[]	**transform**(double[] f, TransformType type) Returns the (forward, inverse) transform of the specified real data set.
double[]	**transform**(UnivariateFunction f, double min, double max, int n, TransformType type) Returns the (forward, inverse) transform of the specified real function, sampled on the specified interval.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

FastSineTransformer

public **FastSineTransformer**(DstNormalization normalization)

Creates a new instance of this class, with various normalization conventions.

Parameters:
normalization - the type of normalization to be applied to the transformed data

Method Detail

transform

public double[] **transform**(double[] f,
TransformType type)

Returns the (forward, inverse) transform of the specified real data set. The first element of the specified data set is required to be 0.

Specified by:
transform in interface RealTransformer
Parameters:
f - the real data array to be transformed (signal)
type - the type of transform (forward, inverse) to be performed
Returns:
the real transformed array (spectrum)
Throws:

MathIllegalArgumentException - if the length of the data array is not a power of two, or the first element of the data array is not zero

---

transform

public double[] **transform**(UnivariateFunction f,
double min,
double max,
int n,
TransformType type)

Returns the (forward, inverse) transform of the specified real function, sampled on the specified interval. This implementation enforces f(x) = 0.0 at x = 0.0.

Specified by:
transform in interface RealTransformer

Parameters:
f - the function to be sampled and transformed
min - the (inclusive) lower bound for the interval
max - the (exclusive) upper bound for the interval
n - the number of sample points
type - the type of transform (forward, inverse) to be performed

Returns:
the real transformed array

Throws:
NonMonotonicSequenceException - if the lower bound is greater than, or equal to the upper bound
NotStrictlyPositiveException - if the number of sample points is negative
MathIllegalArgumentException - if the number of sample points is not a power of two

---

fst

protected double[] **fst**(double[] f)
throws MathIllegalArgumentException

Perform the FST algorithm (including inverse). The first element of the data set is required to be 0.

Parameters:
f - the real data array to be transformed

Returns:
the real transformed array

Throws:
MathIllegalArgumentException - if the length of the data array is not a power of two, or the first element of the data array is not zero

org.apache.commons.math3.transform
Class FastHadamardTransformer

java.lang.Object
  org.apache.commons.math3.transform.FastHadamardTransformer

All Implemented Interfaces:
Serializable, RealTransformer

---

public class **FastHadamardTransformer**extends `Object`implements `RealTransformer`, `Serializable`

Implements the Fast Hadamard Transform (FHT). Transformation of an input vector x to the output vector y.

In addition to transformation of real vectors, the Hadamard transform can transform integer vectors into integer vectors. However, this integer transform cannot be inverted directly. Due to a scaling factor it may lead to rational results. As an example, the inverse transform of integer vector (0, 1, 0, 1) is rational vector (1/2, -1/2, 0, 0).

Since:
2.0

Version:
$Id: FastHadamardTransformer.java 1385310 2012-09-16 16:32:10Z tn $

See Also:
Serialized Form

Constructor Summary	
FastHadamardTransformer()	

Method Summary	
protected double[]	**fht**(double[] x) The FHT (Fast Hadamard Transformation) which uses only subtraction and addition.
protected int[]	**fht**(int[] x) Returns the forward transform of the specified integer data set.
double[]	**transform**(double[] f, TransformType type) Returns the (forward, inverse) transform of the specified real data set.
int[]	**transform**(int[] f) Returns the forward transform of the specified integer data set.The integer transform cannot be inverted directly, due to a scaling factor which may lead to double results.
double[]	**transform**(UnivariateFunction f, double min, double max, int n, TransformType type) Returns the (forward, inverse) transform of the specified real function, sampled on the specified interval.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

FastHadamardTransformer

public **FastHadamardTransformer**()

Method Detail

transform

```
public double[] transform(double[] f,
                          TransformType type)
```

Returns the (forward, inverse) transform of the specified real data set.

Specified by:
    transform in interface RealTransformer

Parameters:
    f - the real data array to be transformed (signal)
    type - the type of transform (forward, inverse) to be performed

Returns:
    the real transformed array (spectrum)

Throws:
    MathIllegalArgumentException - if the length of the data array is not a power of two

---

transform

```
public double[] transform(UnivariateFunction f,
                          double min,
                          double max,
                          int n,
                          TransformType type)
```

Returns the (forward, inverse) transform of the specified real function, sampled on the specified interval.

Specified by:
    transform in interface RealTransformer

Parameters:
    f - the function to be sampled and transformed
    min - the (inclusive) lower bound for the interval
    max - the (exclusive) upper bound for the interval
    n - the number of sample points
    type - the type of transform (forward, inverse) to be performed

Returns:
    the real transformed array

Throws:
    NonMonotonicSequenceException - if the lower bound is greater than, or equal to the upper bound
    NotStrictlyPositiveException - if the number of sample points is negative
    MathIllegalArgumentException - if the number of sample points is not a power of two

transform

public int[] **transform**(int[] f)

Returns the forward transform of the specified integer data set.The integer transform cannot be inverted directly, due to a scaling factor which may lead to double results.

Parameters:
f - the integer data array to be transformed (signal)

Returns:
the integer transformed array (spectrum)

Throws:
MathIllegalArgumentException - if the length of the data array is not a power of two

---

fht

protected double[] **fht**(double[] x)
throws MathIllegalArgumentException

The FHT (Fast Hadamard Transformation) which uses only subtraction and addition. Requires N * log2(N) = n * 2^n additions.

Short Table of manual calculation for N=8

1. **x** is the input vector to be transformed,
2. **y** is the output vector (Fast Hadamard transform of **x**),
3. a and b are helper rows.

x	a	b	y
x_0	$a_0 = x_0 + x_1$	$b_0 = a_0 + a_1$	$y_0 = b_0 + b_1$
x_1	$a_1 = x_2 + x_3$	$b_0 = a_2 + a_3$	$y_0 = b_2 + b_3$
x_2	$a_2 = x_4 + x_5$	$b_0 = a_4 + a_5$	$y_0 = b_4 + b_5$
x_3	$a_3 = x_6 + x_7$	$b_0 = a_6 + a_7$	$y_0 = b_6 + b_7$
x_4	$a_0 = x_0 - x_1$	$b_0 = a_0 - a_1$	$y_0 = b_0 - b_1$
x_5	$a_1 = x_2 - x_3$	$b_0 = a_2 - a_3$	$y_0 = b_2 - b_3$
x_6	$a_2 = x_4 - x_5$	$b_0 = a_4 - a_5$	$y_0 = b_4 - b_5$
x_7	$a_3 = x_6 - x_7$	$b_0 = a_6 - a_7$	$y_0 = b_6 - b_7$

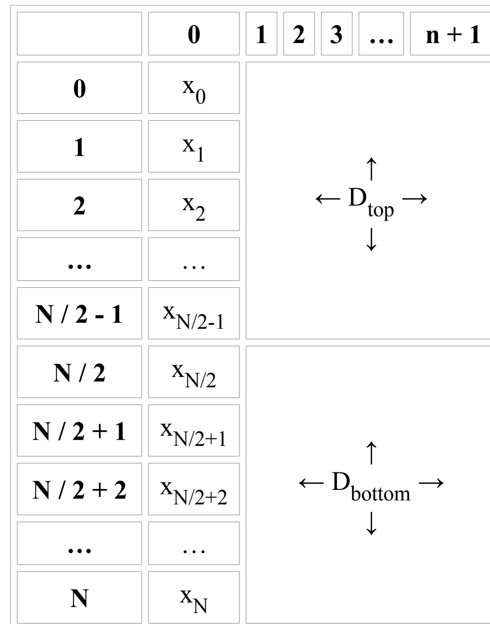
How it works

1. Construct a matrix with N rows and n + 1 columns, hadm[n+1][N].
   (If I use [x][y] it always means [row-offset][column-offset] of a Matrix with n rows and m columns. Its entries go from M[0][0] to M[n][N])
2. Place the input vector x[N] in the first column of the matrix hadm.
3. The entries of the submatrix D_top are calculated as follows
   - D_top goes from entry [0][1] to [N / 2 - 1][n + 1],
   - the columns of D_top are the pairwise mutually exclusive sums of the previous column.
4. The entries of the submatrix D_bottom are calculated as follows

- D_bottom goes from entry [N / 2][1] to [N][n + 1],
- the columns of D_bottom are the pairwise differences of the previous column.

5. The consputation of D_top and D_bottom are best understood with the above example (for N = 8).
6. The output vector y is now in the last column of hadm.
7. Algorithm from *chipcenter*.

Visually



	0	**1**	**2**	**3**	**…**	**n + 1**
0	x_0					
1	x_1					
2	x_2	← D_{top} →				
…	…					
N / 2 - 1	$x_{N/2-1}$					
N / 2	$x_{N/2}$					
N / 2 + 1	$x_{N/2+1}$					
N / 2 + 2	$x_{N/2+2}$	← D_{bottom} →				
…	…					
N	x_N					

Parameters:

x - the real data array to be transformed

Returns:

the real transformed array, y

Throws:

MathIllegalArgumentException - if the length of the data array is not a power of two

---

fht

protected int[] **fht**(int[] x)
throws `MathIllegalArgumentException`

Returns the forward transform of the specified integer data set. The FHT (Fast Hadamard Transform) uses only subtraction and addition.

Parameters:

x - the integer data array to be transformed

Returns:

the integer transformed array, y

Throws:

MathIllegalArgumentException - if the length of the data array is not a power of two

Geometry

The geometry package provides classes useful for many physical simulations in Euclidean spaces, like vectors and rotations in 3D, as well as a general implentation of Binary Space Partitioning Trees (BSP trees).

11.2 Euclidean spaces

Interval and IntervalsSet represent one dimensional regions. All classical set operations are available for intervals sets: union, intersection, symmetric difference (exclusive or), difference, complement, as well as region predicates (point inside/outside/on boundary, emptiness, other region contained). It is also possible to compute geometrical properties like size, barycenter or boundary size. Intervals sets can be built by constructive geometry (union, intersection ...) or from a boundary representation.

PolygonsSet represent two dimensional regions. All classical set operations are available for polygons sets: union, intersection, symmetric difference (exclusive or), difference, complement, as well as region predicates (point inside/outside/on boundary, emptiness, other region contained). It is also possible to compute geometrical properties like size, barycenter or boundary size and to extract the vertices. Polygons sets can be built by constructive geometry (union, intersection ...) or from a boundary representation.

PolyhedronsSet represent three dimensional regions. All classical set operations are available for polyhedrons sets: union, intersection, symmetric difference (exclusive or), difference, complement, as well as region predicates (point inside/outside/on boundary, emptiness, other region contained). It is also possible to compute geometrical properties like size, barycenter or boundary size and to extract the vertices. Polyhedrons sets can be built by constructive geometry (union, intersection ...) or from a boundary representation.

Vector3D provides a simple vector type. One important feature is that instances of this class are guaranteed to be immutable, this greatly simplifies modelling dynamical systems with changing states: once a vector has been computed, a reference to it is known to preserve its state as long as the reference itself is preserved.

Numerous constructors are available to create vectors. In addition to the straightforward cartesian coordinates constructor, a constructor using azimuthal coordinates can build normalized vectors and linear constructors from one, two, three or four base vectors are also available. Constants have been defined for the most commons vectors (plus and minus canonical axes, null vector, and special vectors with infinite or NaN coordinates).

The generic vectorial space operations are available including dot product, normalization, orthogonal vector finding and angular separation computation which have a specific meaning in 3D. The 3D geometry specific cross product is of course also implemented.

Vector3DFormat is a specialized format for formatting output or parsing input with text representation of 3D vectors.

Rotation represents 3D rotations. Rotation instances are also immutable objects, as Vector3D instances.

Rotations can be represented by several different mathematical entities (matrices, axe and angle, Cardan or Euler angles, quaternions). This class presents a higher level abstraction, more user-oriented and hiding implementation details. Well, for the curious, we use quaternions for the internal representation. The user can build a rotation from any of these representations, and any of these representations can be retrieved from aRotation instance (see the various constructors and getters). In addition, a rotation can also be built implicitely from a set of vectors and their image.

This implies that this class can be used to convert from one representation to another one. For example, converting a rotation matrix into a set of Cardan angles can be done using the following single line of code:

```
double[] angles = new Rotation(matrix, 1.0e-10).getAngles(RotationOrder.XYZ);
```

Focus is oriented on what a rotation *does* rather than on its underlying representation. Once it has been built, and regardless of its internal representation, a rotation is an*operator* which basically transforms three dimensional vectors into other three dimensional vectors. Depending on the application, the meaning of these vectors may vary as well as the semantics of the rotation.

For example in a spacecraft attitude simulation tool, users will often consider the vectors are fixed (say the Earth direction for example) and the rotation transforms the coordinates coordinates of this vector in inertial frame into the coordinates of the same vector in satellite frame. In this case, the rotation implicitly defines the relation between the two frames (we have fixed vectors and moving frame). Another example could be a telescope control application, where the rotation would transform the sighting direction at rest into the desired observing direction when the telescope is pointed towards an object of interest. In this case the rotation transforms the direction at rest in a topocentric frame into the sighting direction in the same topocentric frame (we have moving vectors in fixed frame). In many case, both approaches will be combined, in our telescope example, we will probably also need to transform the observing direction in the topocentric frame into the observing direction in inertial frame taking into account the observatory location and the Earth rotation.

These examples show that a rotation means what the user wants it to mean, so this class does not push the user towards one specific definition and hence does not provide methods like projectVectorIntoDestinationFrame or computeTransformedDirection. It provides simpler and more generic methods: applyTo(Vector3D) andapplyInverseTo(Vector3D).

Since a rotation is basically a vectorial operator, several rotations can be composed together and the composite operation $r = r_1$ o r_2 (which means that for each vectoru, $r(u) = r_1(r_2(u))$) is also a rotation. Hence we can consider that in addition to vectors, a rotation can be applied to other rotations as well (or to itself). With our previous notations, we would say we can apply r_1 to r_2 and the result we get is $r = r_1$o r_2. For this purpose, the class provides the methods: applyTo(Rotation) andapplyInverseTo(Rotation).

11.3 Binary Space Partitioning

BSP trees are an efficient way to represent space partitions and to associate attributes with each cell. Each node in a BSP tree represents a convex region which is partitioned in two convex sub-regions at each side of a cut hyperplane. The root tree contains the complete space.

The main use of such partitions is to use a boolean attribute to define an inside/outside property, hence representing arbitrary polytopes (line segments in 1D, polygons in 2D and polyhedrons in 3D) and to operate on them.

Another example would be to represent Voronoi tesselations, the attribute of each cell holding the defining point of the cell.

The application-defined attributes are shared among copied instances and propagated to split parts. These attributes are not used by the BSP-tree algorithms themselves, so the application can use them for any purpose. Since the tree visiting method holds internal and leaf nodes differently, it is possible to use different classes for internal nodes attributes and leaf nodes attributes. This should be used with care, though, because if the tree is modified in any way after attributes have been set, some internal nodes may become leaf nodes and some leaf nodes may become internal nodes.

Optimization

The optimization package provides algorithms to optimize (i.e. either minimize or maximize) some objective or cost function. The package is split in several sub-packages dedicated to different kind of functions or algorithms.

- the univariate package handles univariate scalar functions,
- the linear package handles multivariate vector linear functions with linear constraints,
- the direct package handles multivariate scalar functions using direct search methods (i.e. not using derivatives),
- the general package handles multivariate scalar or vector functions using derivatives.
- the fitting package handles curve fitting by univariate real functions

The top level optimization package provides common interfaces for the optimization algorithms provided in sub-packages. The main interfaces defines defines optimizers and convergence checkers. The functions that are optimized by the algorithms provided by this package and its sub-packages are a subset of the one defined in the analysispackage, namely the real and vector valued functions. These functions are called objective function here. When the goal is to minimize, the functions are often called cost function, this name is not used in this package.

The type of goal, i.e. minimization or maximization, is defined by the enumeratedGoalType which has only two values: MAXIMIZE and MINIMIZE.

Optimizers are the algorithms that will either minimize or maximize, the objective function by changing its input variables set until an optimal set is found. There are only four interfaces defining the common behavior of optimizers, one for each supported type of objective function:

- UnivariateOptimizer for univariate real functions
- MultivariateOptimizer for multivariate real functions
- DifferentiableMultivariateOptimizer for differentiable multivariate real functions
- DifferentiableMultivariateVectorOptimizer for differentiable multivariate vectorial functions

Despite there are only four types of supported optimizers, it is possible to optimize a transform a non-differentiable multivariate vectorial function by converting it to a non-differentiable multivariate real function thanks to the LeastSquaresConverter helper class. The transformed function can be optimized using any implementation of theMultivariateOptimizer interface.

For each of the four types of supported optimizers, there is a special implementation which wraps a classical optimizer in order to add it a multi-start feature. This feature call the underlying optimizer several times in sequence with different starting points and returns the best optimum found or all optima if desired. This is a classical way to prevent being trapped into a local extremum when looking for a global one.

12.2 Univariate Functions

A UnivariateOptimizer is used to find the minimal values of a univariate real-valued function f.

These algorithms usage is very similar to root-finding algorithms usage explained in the analysis package. The main difference is that the solve methods in root finding algorithms is replaced by optimize methods.

12.3 Linear Programming

This package provides an implementation of George Dantzig's simplex algorithm for solving linear optimization problems with linear equality and inequality constraints.

12.4 Direct Methods

Direct search methods only use cost function values, they don't need derivatives and don't either try to compute approximation of the derivatives. According to a 1996 paper by Margaret H. Wright (Direct Search Methods: Once Scorned, Now Respectable), they are used when either the computation of the derivative is impossible (noisy functions, unpredictable discontinuities) or difficult (complexity, computation cost). In the first cases, rather than an optimum, a *not too bad* point is desired. In the latter cases, an optimum is desired but cannot be reasonably found. In all cases direct search methods can be useful.

Simplex-based direct search methods are based on comparison of the cost function values at the vertices of a simplex (which is a set of n+1 points in dimension n) that is updated by the algorithms steps.

The instances can be built either in single-start or in multi-start mode. Multi-start is a traditional way to try to avoid being trapped in a local minimum and miss the global minimum of a function. It can also be used to verify the convergence of an algorithm. In multi-start mode, the minimizesmethod returns the best minimum found after all starts, and the etMinima method can be used to retrieve all minima from all starts (including the one already provided by the minimizes method).

The direct package provides four solvers:

- the classical Nelder-Mead method,
- Virginia Torczon's multi-directional method,
- Nikolaus Hansen's Covariance Matrix Adaptation Evolution Strategy (CMA-ES),
- Mike Powell's BOBYQA method.

The first two simplex-based methods do not handle simple bounds constraints by themselves. However there are two adapters(MultivariateFunctionMappingAdapter andMultivariateFunctionPenaltyAdapter) that can be used to wrap the user function in such a way the wrapped function is unbounded and can be used with these optimizers, despite the fact the underlying function is still bounded and will be called only with feasible points that fulfill the constraints. Note however that using these adapters are only a poor man solutions to simple bounds optimization constraints. Better solutions are to use an optimizer that directly supports simple bounds. Some caveats of the mapping adapter solution are that

- behavior near the bounds may be numerically unstable as bounds are mapped from infinite values,
- start value is evaluated by the optimizer as an unbounded variable, so it must be converted from bounded to unbounded by user,
- optimum result is evaluated by the optimizer as an unbounded variable, so it must be converted from unbounded to bounded by user,
- convergence values are evaluated by the optimizer as unbounded variables, so there will be scales differences when converted to bounded variables,
- in the case of simplex based solvers, the initial simplex should be set up as delta in unbounded variables.

One caveat of penalty adapter is that if start point or start simplex is outside of the allowed range, only the penalty function is used, and the optimizer may converge without ever entering the allowed range.

The last methods do handle simple bounds constraints directly, so the adapters are not needed with them.

12.5 General Case

The general package deals with non-linear vectorial optimization problems when the partial derivatives of the objective function are available.

One important class of estimation problems is weighted least squares problems. They basically consist in finding the values for some parameters p_k such that a cost function $J = sum(w_i(mes_i - mod_i)^2)$ is minimized.

The various ($target_i$ - $model_i(p_k)$) terms are called residuals. They represent the deviation between a set of target values $target_i$ and theoretical values computed from models $model_i$ depending on free parameters p_k. The w_i factors are weights. One classical use case is when the target values are experimental observations or measurements.

Solving a least-squares problem is finding the free parameters p_k of the theoretical models such that they are close to the target values, i.e. when the residual are small.

Two optimizers are available in the general package, both devoted to least-squares problems. The first one is based on the Gauss-Newton method. The second one is theLevenberg-Marquardt method.

In order to solve a vectorial optimization problem, the user must provide it as an object implementing the DifferentiableMultivariateVectorFunction interface. The object will be provided to the estimate method of the optimizer, along with the target and weight arrays, thus allowing the optimizer to compute the residuals at will. The last parameter to the estimate method is the point from which the optimizer will start its search for the optimal point.

Quadratic Problem Example

We are looking to find the best parameters [a, b, c] for the quadratic function **f(x) = a x^2 + b x + c**. The data set below was generated using [a = 8, b = 10, c = 16]. A random number between zero and one was added to each y value calculated.

X	Y
1	34.234064369
2	68.2681162306108
3	118.615899084602
4	184.138197238557
5	266.599877916276
6	364.147735251579
7	478.019226091914
8	608.140949270688
9	754.598868667148
10	916.128818085883

First we need to implement the interface DifferentiableMultivariateVectorFunction. This requires the implementation of the method signatures:

- **MultivariateMatrixFunction jacobian()**
- **double[] value(double[] point)**

We'll tackle the implementation of the MultivariateMatrixFunction jacobian()method first. You may wish to familiarize yourself with what a Jacobian Matrix is. In this case the Jacobian is the partial derivative of the function with respect to the parameters a, b and c. These derivatives are computed as follows:

- $d(ax^2 + bx + c)/da = x^2$
- $d(ax^2 + bx + c)/db = x$
- $d(ax^2 + bx + c)/dc = 1$

For a quadratic which has three variables the Jacobian Matrix will have three columns, one for each variable, and the number of rows will equal the number of rows in our data set, which in this case is ten. So for example for [a = 1, b = 1, c = 1], the Jacobian Matrix is (excluding the first column which shows the value of x):

x	$d(ax^2 + bx + c)/da$	$d(ax^2 + bx + c)/db$	$d(ax^2 + bx + c)/dc$
1	1	1	1
2	4	2	1
3	9	3	1
4	16	4	1
5	25	5	1
6	36	6	1
7	49	7	1
8	64	8	1
9	81	9	1
10	100	10	1

The implementation of the MultivariateMatrixFunction jacobian() for this problem looks like this (The x parameter is an ArrayList containing the independent values of the data set):

```
private double[][] jacobian(double[] variables) {
    double[][] jacobian = new double[x.size()][3];
    for (int i = 0; i < jacobian.length; ++i) {
        jacobian[i][0] = x.get(i) * x.get(i);
        jacobian[i][1] = x.get(i);
        jacobian[i][2] = 1.0;
    }
    return jacobian;
}
public MultivariateMatrixFunction jacobian() {
    return new MultivariateMatrixFunction() {
        private static final long serialVersionUID = -8673650298627399464L;
        public double[][] value(double[] point) {
            return jacobian(point);
        }
    };
}
```

Note that if for some reason the derivative of the objective function with respect to its variables is difficult to obtain, Numerical differentiation can be used.

The implementation of the double[] value(double[] point) method, which returns a double array containing the values the objective function returns per given independent value and the current set of variables or parameters, can be seen below:

```
public double[] value(double[] variables) {
    double[] values = new double[x.size()];
    for (int i = 0; i < values.length; ++i) {
        values[i] = (variables[0] * x.get(i) + variables[1]) * x.get(i) + variables[2];
    }
    return values;
}
```

Below is the the class containing all the implementation details (Taken from the Apache Commons Math**org.apache.commons.math3.optimization.general.LevenbergMarquardtOptimizerTest**):

```
private static class QuadraticProblem
    implements DifferentiableMultivariateVectorFunction, Serializable {
    private static final long serialVersionUID = 7072187082052755854L;
    private List<Double> x;
    private List<Double> y;
    public QuadraticProblem() {
```

```
        x = new ArrayList<Double>();
        y = new ArrayList<Double>();
    }
    public void addPoint(double x, double y) {
        this.x.add(x);
        this.y.add(y);
    }
    public double[] calculateTarget() {
        double[] target = new double[y.size()];
        for (int i = 0; i < y.size(); i++) {
                target[i] = y.get(i).doubleValue();
        }
        return target;
    }
    private double[][] jacobian(double[] variables) {
        double[][] jacobian = new double[x.size()][3];
        for (int i = 0; i < jacobian.length; ++i) {
            jacobian[i][0] = x.get(i) * x.get(i);
            jacobian[i][1] = x.get(i);
            jacobian[i][2] = 1.0;
        }
        return jacobian;
    }
    public double[] value(double[] variables) {
        double[] values = new double[x.size()];
        for (int i = 0; i < values.length; ++i) {
            values[i] = (variables[0] * x.get(i) + variables[1]) * x.get(i) + variables[2];
        }
        return values;
    }
    public MultivariateMatrixFunction jacobian() {
        return new MultivariateMatrixFunction() {
            private static final long serialVersionUID = -8673650298627399464L;
            public double[][] value(double[] point) {
                return jacobian(point);
            }
        };
    }
}
```

The below code shows how to go about using the above class and a LevenbergMarquardtOptimizer instance to produce an optimal set of quadratic curve fitting parameters:

```
QuadraticProblem problem = new QuadraticProblem();
problem.addPoint(1, 34.234064369);
problem.addPoint(2, 68.2681162306);
problem.addPoint(3, 118.6158990846);
problem.addPoint(4, 184.1381972386);
problem.addPoint(5, 266.5998779163);
problem.addPoint(6, 364.1477352516);
problem.addPoint(7, 478.0192260919);
problem.addPoint(8, 608.1409492707);
problem.addPoint(9, 754.5988686671);
problem.addPoint(10, 916.1288180859);
LevenbergMarquardtOptimizer optimizer = new LevenbergMarquardtOptimizer();
final double[] weights = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };
final double[] initialSolution = {1, 1, 1};
PointVectorValuePair optimum = optimizer.optimize(100,
                                        problem,
```

```
                                      problem.calculateTarget(),
                                      weights,
                                      initialSolution);
final double[] optimalValues = optimum.getPoint();
System.out.println("A: " + optimalValues[0]);
System.out.println("B: " + optimalValues[1]);
System.out.println("C: " + optimalValues[2]);
```

If you run the above sample you will see the following printed by the console:

```
A: 7.998832172372726
B: 10.001841530162448
C: 16.324008168386605
```

In addition to least squares solving, the NonLinearConjugateGradientOptimizer class provides a non-linear conjugate gradient algorithm to optimizeDifferentiableMultivariateFunction. Both the Fletcher-Reeves and the Polak-Ribière search direction update methods are supported. It is also possible to set up a preconditioner or to change the line-search algorithm of the inner loop if desired (the default one is a Brent solver).

The PowellOptimizer provides an optimization method for non-differentiable functions.

12.6 Curve Fitting

The fitting package deals with curve fitting for univariate real functions. When a univariate real function y = f(x) does depend on some unknown parameters p_0, p_1 ... p_{n-1}, curve fitting can be used to find these parameters. It does this by *fitting* the curve so it remains very close to a set of observed points (x_0, y_0), (x_1, y_1) ... (x_{k-1}, y_{k-1}). This fitting is done by finding the parameters values that minimizes the objective function $sum(y_i-f(x_i))^2$. This is really a least squares problem.

For all provided curve fitters, the operating principle is the same. Users must first create an instance of the fitter, then add the observed points and once the complete sample of observed points has been added they must call the fit method which will compute the parameters that best fit the sample. A weight is associated with each observed point, this allows to take into account uncertainty on some points when they come from loosy measurements for example. If no such information exist and all points should be treated the same, it is safe to put 1.0 as the weight for all points.

The CurveFitter class provides curve fitting for general curves. Users must provide their own implementation of the curve template as a class implementing theParametricUnivariateFunction interface and they must provide the initial guess of the parameters.

The following example shows how to fit data with a polynomial function.

```
final CurveFitter fitter = new CurveFitter(new LevenbergMarquardtOptimizer());
fitter.addObservedPoint(-1.00, 2.021170021833143);
fitter.addObservedPoint(-0.99, 2.221135431136975);
fitter.addObservedPoint(-0.98, 2.09985277659314);
fitter.addObservedPoint(-0.97, 2.0211192647627025);
// ... Lots of lines omitted ...
fitter.addObservedPoint( 0.99, -2.4345814727089854);
// The degree of the polynomial is deduced from the length of the array containing
// the initial guess for the coefficients of the polynomial.
final double[] init = { 12.9, -3.4, 2.1 }; // 12.9 - 3.4 x + 2.1 x^2
// Compute optimal coefficients.
final double[] best = fitter.fit(new PolynomialFunction.Parametric(), init);
// Construct the polynomial that best fits the data.
final PolynomialFunction fitted = new PolynomialFunction(best);
```

The more specialized HarmonicFitter classes requires neither an implementation of the parametric real function nor an initial guess as it is are able to compute them internally.

Ordinary Differential Equations Integration

13.1 Overview

The ode package provides classes to solve Ordinary Differential Equations problems.

This package solves Initial Value Problems of the form y'=f(t,y) with t_0 and $y(t_0)=y_0$known. The provided integrators compute an estimate of y(t) from $t=t_0$ to $t=t_1$.

All 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 theStepInterpolator 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 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 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 his problem in his own classes which should implement theFirstOrderDifferentialEquations interface. Then he should pass it to the integrator he prefers among all the classes that implement the FirstOrderIntegrator interface. The following example shows how to implement the simple two-dimensional problem:

- $y'_0(t) = \omega \times (c_1 - y_1(t))$
- $y'_1(t) = \omega \times (y_0(t) - c_0)$

with some initial state $y(t_0) = (y_0(t_0), y_1(t_0))$. In fact, the exact solution of this problem is that y(t) moves along a circle centered at $c = (c_0, c_1)$ with constant angular rate ω.

```
private static class CircleODE implements FirstOrderDifferentialEquations {
    private double[] c;
    private double omega;
    public CircleODE(double[] c, double omega) {
        this.c     = c;
        this.omega = omega;
    }
    public int getDimension() {
        return 2;
    }
    public void computeDerivatives(double t, double[] y, double[] yDot) {
        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 Dormand-Prince 8(5,3) integrator as an example):

```
FirstOrderIntegrator dp853 = new DormandPrince853Integrator(1.0e-8, 100.0, 1.0e-10, 1.0e-10);
FirstOrderDifferentialEquations ode = new CircleODE(new double[] { 1.0, 1.0 }, 0.1);
double[] y = new double[] { 0.0, 1.0 }; // initial state
dp853.integrate(ode, 0.0, y, 16.0, y); // now y contains final state at time t=16.0
```

13.2 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 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 theStepHandler interface or a StepNormalizer object wrapping a user-specified object implementing the FixedStepHandler interface into the integrator before calling theFirstOrderIntegrator.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 it really is a circle arc. We simply add the following before the call to integrator.integrate:

```
StepHandler stepHandler = new StepHandler() {
    public void init(double t0, double[] y0, double t) {
    }

    public void handleStep(StepInterpolator interpolator, boolean isLast) {
        double   t = interpolator.getCurrentTime();
        double[] y = interpolator.getInterpolatedState();
        System.out.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 Serializableinterface. 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. TheStepNormalizer 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 he prefers to retain full control over the integration or if the automatic guess is wrong.

13.3 Discrete Events Handling

ODE problems are continuous ones. However, sometimes discrete events must be taken into account. The most frequent case is 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. For the sake of root finding, it should however be continuous (but not necessarily smooth) at least in the roots vicinity. The steps are shortened as needed to ensure the events occur at step boundaries (even if the integrator is a fixed-step integrator).

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 to him:

- integration can be stopped (this is called a G-stop facility),
- the state vector or the derivatives can be changed,
- 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 target state is reached, 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:

```
public double g(double t, double[] y) {
  return y[0] - targetConcentration;
}
public int eventOccurred(double t, double[] y, boolean increasing) {
  return STOP;
}
```

The second case, change 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:

```
public double g(double t, double[] y) {
  return (t - tManeuverStart) * (t - tManeuverStop);
}
public int eventOccurred(double t, double[] y, boolean increasing) {
  return RESET_DERIVATIVES;
}
```

The third case is useful mainly for monitoring purposes, a simple example is:

```
public double g(double t, double[] y) {
  return y[0] - y[1];
}
public int eventOccurred(double t, double[] y, boolean increasing) {
  logger.log("y0(t) and y1(t) curves cross at t = " + t);
  return CONTINUE;
}
```

13.4 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 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

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

13.5 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, he will 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 user.

This behavior imply 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, he will use the specialized step handler and event handler classes registered at top level. He 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 a 60 elements array. 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 he will implement the ParameterizedODE interface. Considering again our example where only ω is considered a parameter, we get:

```
public class BasicCircleODE implements ParameterizedODE {
    private double[] c;
    private double omega;
    public BasicCircleODE(double[] c, double omega) {
        this.c     = c;
        this.omega = omega;
    }
    public int getDimension() {
        return 2;
    }
    public void computeDerivatives(double t, double[] y, double[] yDot) {
        yDot[0] = omega * (c[1] - y[1]);
        yDot[1] = omega * (y[0] - c[0]);
    }
    public int getParametersDimension() {
        // we are only interested in the omega parameter
        return 1;
    }
    public void setParameter(int i, double value) {
        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):

```
double[] hY = new double[] { 0.001, 0.001 };
double[] hP = new double[] { 1.0e-6 };
FirstOrderIntegratorWithJacobians integrator = 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 by himself. So rather than the FirstOrderDifferentialEquations interface he will implement the ODEWithJacobians interface. Considering again our example where only ω is considered a parameter, we get:

```
public class EnhancedCircleODE implements ODEWithJacobians {
    private double[] c;
    private double omega;
    public EnhancedCircleODE(double[] c, double omega) {
        this.c     = c;
        this.omega = omega;
    }
    public int getDimension() {
        return 2;
    }
    public void computeDerivatives(double t, double[] y, double[] yDot) {
        yDot[0] = omega * (c[1] - y[1]);
        yDot[1] = omega * (y[0] - c[0]);
    }
    public int getParametersDimension() {
        // we are only interested in the omega parameter
        return 1;
```

```
    }
    public void computeJacobians(double t, double[] y, double[] yDot, double[][] dFdY, double[][] dFdP) {
        dFdY[0][0] = 0;
        dFdY[0][1] = -omega;
        dFdY[1][0] = omega;
        dFdY[1][1] = 0;
        dFdP[0][0] = 0;
        dFdP[0][1] = omega;
        dFdP[0][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:

```
FirstOrderIntegratorWithJacobians integrator = new FirstOrderIntegratorWithJacobians(dp853, ode);
integrator.integrate(t0, y0, dy0dp, t, y, dydy0, dydp);
```

Genetic Algorithms

14.1 Overview

The genetics package provides a framework and implementations for genetic algorithms.

14.2 GA Framework

GeneticAlgorithm provides an execution framework for Genetic Algorithms (GA).Populations, consisting of Chromosomes are evolved by the GeneticAlgorithm until aStoppingCondition is reached. Evolution is determined by SelectionPolicy, MutationPolicyand Fitness.

The GA itself is implemented by the evolve method of the GeneticAlgorithm class, which looks like this:

```
public Population evolve(Population initial, StoppingCondition condition) {
  Population current = initial;
  while (!condition.isSatisfied(current)) {
    current = nextGeneration(current);
  }
  return current;
}
```

The nextGeneration method implements the following algorithm:

1. Get nextGeneration population to fill from current generation, using its nextGeneration method
2. Loop until new generation is filled:
   - Apply configured SelectionPolicy to select a pair of parents from current
   - With probability = getCrossoverRate(), apply configured CrossoverPolicy to parents
   - With probability = getMutationRate(), apply configured MutationPolicy to each of the offspring
   - Add offspring individually to nextGeneration, space permitting
3. Return nextGeneration

14.3 Implementation

Here is an example GA execution:

```
// initialize a new genetic algorithm
GeneticAlgorithm ga = new GeneticAlgorithm(
  new OnePointCrossover<Integer>(),
  1,
  new RandomKeyMutation(),
  0.10,
  new TournamentSelection(TOURNAMENT_ARITY)
);

// initial population
Population initial = getInitialPopulation();

// stopping condition
StoppingCondition stopCond = new FixedGenerationCount(NUM_GENERATIONS);
```

```
// run the algorithm
Population finalPopulation = ga.evolve(initial, stopCond);

// best chromosome from the final population
Chromosome bestFinal = finalPopulation.getFittestChromosome();
```

The arguments to the GeneticAlgorithm constructor above are:

Parameter	value in example	meaning
crossoverPolicy	OnePointCrossover	A random crossover point is selected and the first part from each parent is copied to the corresponding child, and the second parts are copied crosswise.
crossoverRate	1	Always apply crossover
mutationPolicy	RandomKeyMutation	Changes a randomly chosen element of the array representation to a random value uniformly distributed in [0,1].
mutationRate	0,1	Apply mutation with probability 0.1 - that is, 10% of the time.
selectionPolicy	TournamentSelection	Each of the two selected chromosomes is selected based on an n-ary tournament -- this is done by drawing n random chromosomes without replacement from the population, and then selecting the fittest chromosome among them.

The algorithm starts with an initial population of Chromosomes. and executes until the specified StoppingCondition is reached. In the example above, a FixedGenerationCountstopping condition is used, which means the algorithm proceeds through a fixed number of generations.

Curve Fitting

17.1 Overview

The fitting package deals with curve fitting for univariate real functions. When a univariate real function y = f(x) does depend on some unknown parameters p_0, p_1 ... p_{n-1}, curve fitting can be used to find these parameters. It does this by *fitting* the curve so it remains very close to a set of observed points (x_0, y_0), (x_1, y_1) ... (x_{k-1}, y_{k-1}). This fitting is done by finding the parameters values that minimizes the objective function $\Sigma(y_i - f(x_i))^2$. This is actually a **least-squares** problem.

For all provided curve fitters, the operating principle is the same. Users must first **create an instance of the fitter**, then **add the observed points** and once the complete sample of observed points has been added they must call **the fit method which will compute the parameters that best fit the sample**. A weight is associated with each observed point, this allows to take into account uncertainty on some points when they come from loosy measurements for example. If no such information exist and all points should be treated the same, it is safe to put 1.0 as the weight for all points.

17.2 General Case

The CurveFitter class provides curve fitting for general curves. Users must provide their own implementation of the curve template as a class implementing the ParametricUnivariateFunctioninterface and they must provide the initial guess of the parameters.

The following example shows how to fit data with a polynomial function.

```
import
org.apache.commons.math3.optim.nonlinear.vector.jacobian.LevenbergMarquar
dtOptimizer

import org.apache.commons.math3.analysis.ParametricUnivariateFunction

import org.apache.commons.math3.util.FastMath

import org.apache.commons.math3.fitting.*

import org.apache.commons.math3.analysis.polynomials.PolynomialFunction


CurveFitter fitter = new CurveFitter(new LevenbergMarquardtOptimizer())

fitter.addObservedPoint(-1.00, 2.021170021833143)

fitter.addObservedPoint(-0.99, 2.221135431136975)

fitter.addObservedPoint(-0.98, 2.09985277659314)

fitter.addObservedPoint(-0.97, 2.0211192647627025)

// ... Lots of lines omitted ...

fitter.addObservedPoint( 0.99, -2.4345814727089854)
```

```
// The degree of the polynomial is deduced from the length of the array containing
// the initial guess for the coefficients of the polynomial.
init = [ 12.9, -3.4, 2.1 ] as  double [] // 12.9 - 3.4 x + 2.1 x^2

// Compute optimal coefficients.
best = fitter.fit(new PolynomialFunction.Parametric(), init)

// Construct the polynomial that best fits the data.
fitted = new PolynomialFunction(best)
```

7.3 Special Cases

There are more specialized classes, for which the appropriate parametric function is implicitly used:

- PolynomialFitter fits a polynomial function
- HarmonicFitter fits a harmonic function
- GaussianFitter fits a Gaussian function

The HarmonicFitter and GaussianFitter also provide a no-argument fit() method that will internally estimate initial guess values for the parameters.