The basic Matrix-like classes of ScalaLab

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

RichDouble2DArray: The *RichDouble2DArray class* wraps the Java/Scala 2-D double arrays and is one of the more important and effective matrix type in ScalaLab. It does not concentrate on a specific library, instead it aims to **exploit the best algorithms in its numerical operations.** Since it is designed to be the most effective type, the static routines in their direct format return a *RichDouble2DArray* object, i.e.

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

Zero-indexed Matrix Libraries

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

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

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

One-indexed Matrix Library

The one-indexed Matrix library wraps the NUMAL library. By convention routines ending operate with that Matrixes, e.g.

var mra1 = rand1(2,3)

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

var r1d = rand(9) // RichDouble1DArray

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

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

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

var vrnd = vrand(9) // vector

Important Conventions for operating effectively with different matrix types

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

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

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

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

Description of the RichDouble2DArray class

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

Constructors

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

var m=new RichDouble2DArray(2, 3)

Construction by specifying the initial elements, e.g.

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

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

Another type of construction:

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

// construct RichDouble2DArray by copying the array values

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

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

dd(1)(1)=11

var mdd = new RichDouble2DArray(dd)

Basic Methods

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

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

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

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

Conversion Routines

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Routines

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

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

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

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

Transpose operator ~

transposes the RichDouble2DArray , e.g.

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

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

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

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

Creating arrays initialized with values

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

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

or as:

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

Similarly for 2-D Arrays:

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

or as:

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

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

Operations on arrays

Let create a 2-D array:

var A = Rand(8, 12)

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

def inc5(x: Double) = x+5

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

var A5 = A map inc5 // map a function

We negate the array simply as:

var A5m = -A5 // unary minus

We multiply all the elements with 10.

*var A10 = 10*A*

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

*var A20 = A*20.0*

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

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

var At = A~

We can multiply arrays as usual:

*var aa = A*At*

We can express convenient expressions, e.g.:

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

The RichDouble1DArray (1-D Double Arrays)

The *RichDouble1DArray* object actually wraps an Array[Double] of Scala or a double[] of Java. However, it presents a large number of *static methods* to the interpreter in order to facilitate the work of the user. Such methods allow more convenient handling with overloading the basic mathematical operators, and using the basic mathematical functions (e.g. sin, cos, tan, etc.) directly on RichDouble1DArray objects.

We present these operations by means of examples.

Operations on 1-D Double Arrays

Construction from elements:

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

or

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

Construction from elements 1-indexed array:

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

Variance of an array:

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

Standard Deviation:

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

Covariance:

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

Correlation:

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

The objective of the RichDouble1DArray class is to provide convenient operations on Java/Scala 1-D arrays

of doubles. Static operations like for example *sin, cos, tan, log, ceil* etc are provided with the *RichDouble1DArray* class. These operations does not convert implicitly the *Array[Double]* type, for example:

var x = new Array[Double](20)
x(2) = 6
var y = sin(x) // sin is provided with the RichDouble1DArray class, and returns also an Array[Double] object

However, we cannot provide convenient operations, like addition, multiplication etc. on an Array[Double] object, since it is a predefined type. Therefore, we exploit the implicit conversions machinery of Scala by means of the conversion to the RichDouble1DArray type. For example:
var x = new Array[Double](20)
x(2) = 6
var y = sin(x)+3 // sin is provided with the RichDouble1DArray class, and returns also an Array[Double] object

The RichDouble2DArray

The RichDouble2DArray class also provides operations on two-dimensional array of doubles, corresponding to the Scala type Array[Array[Double]], and interoperable with the double[][] type of Java. A large number of operations is supported on this class although the syntax is not as elegant as the other classes. Actually, all the routines described in the book "A Numerical Library in Java for Scientists and Engineers" [11] can be used, since 2D-Double Scala arrays are also Java arrays.

The Vec class

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

Constructors

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

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

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

var da = new Array[Double](20)

da(3)=3

var dav = new Vec(da)

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

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

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

Basic Methods

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

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

var l = dav.size

Vectors are **dynamically resizable** while double[] arrays are not. An attempt to update a vector element outside its current range increases automatically the vector's size by a multiplicative factor, e.g. 1.5, in order to avoid frequent resizing operations (which is somehow costly). For example we can execute:

vv(15) = 8

After that, the length of the vector vv will be 22.

Displaying the whole vector contents

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

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

v.print

In-Place operators

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

var v = vrand(100)

var vo = vones(v.length)

v++vo // adds to v the vo vector

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

v--v10 // subtract in-place

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

Scalar append Vector operator ::

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

var vo = vones(4)

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

Scalar prepend Vector operator :::

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

var vo = vones(4)

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

Vector append Scalar operator ::<

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

var vo = vones(4)

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

Vector prepend Scalar operator :::<

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

var vo = vones(4)

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

Vector append Vector operator :::

appends the first vector at the start of the second, e.g.

vones(2) ::: vrand(5)

inc(x1, dx, x2): implements colon operator of MATLAB, i.e. x1:dx:x2

var (x1, dx, x2) = (-12, 0.01, 18)

var x = inc(x1, dx, x2)

*var y = sin(0.12*x)+2.3*cos(1.23*x)*

plot(x,y)

linspace: Linearly spaced vector.

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

```
var N = pow(2, 14).asInstanceOf[Int] // signal length
var t = linspace(0, 10, N) // sample time axis
var F1=30; var F2 = 4; // signal frequencies
var x = sin(F1*t)+2.4*cos(F2*t) // construct synthetic signal
var fftx = fft(x) // perform FFT

def getReal(x: org.apache.commons.math.complex.Complex) = x.getReal
var fftxReals = fftx map getReal
linePlotsOn
figure(1); subplot(2,1,1); plot(x); title("the signal");
subplot(2,1,2); plot(fftxReals);  title("FFT of the signal - real parts")
```

logspace: Logarithmically spaced vector.

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

e.g.

```
var x = logspace(1, 4, 100, 10)
var N = pow(2, 14).toInt  // signal length
var tdefault = logspace(0,10, N)  // default base 10
var base = 2
var t = logspace(0, 10, N, base) // sample time axis
base = 10
var t10 = logspace(0, 10, N, base) // sample time axis
```

dot : Vector dot product, e.g.

```
var rv1 = vrand(1000)  // a random vector
rv1 = rv1-mean(rv1)  // mean subtract
var rv2 = vrand(1000) // another random vector
```

rv2 = rv2-mean(rv2)

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

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

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

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

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

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

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

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

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

var v = vrand(2000)

plot(v)

title("2000 random values")

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

var v = vones(300)

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

var v = vzeros(300)

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

var v = vones(20)

var sm = sum(v)

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

var v = vrand(200)

var mn = mean(v)

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

var v = vfill(300, -0.24)

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

```
var t=inc(-0.9, 0.01, 0.9)
var x = tanh(4*t)
figure(1); subplot(3,2,1); plot(t,x);
var xasin = asin(t); subplot(3,2,2); plot(t, xasin);
var xacos = acos(t); subplot(3,2,3); plot(t, xacos);
var xsinh = sinh(t); subplot(3,2,4); plot(t, xsinh);
var xcosh = cosh(t); subplot(3,2,5); plot(t, xcosh);
var xcomp = 4*cos(4.5*t)+0.34*sin(-18.364*t); subplot(3,2,6); plot(t, xcomp);
```

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

fft: Compute Fast Fourier Transform

```
var F1=30; var F2 = 4; // signal frequencies
var t = linspace(0, 10, 2^^^12)
var x = sin(F1*t)+2.4*cos(F2*t) // construct synthetic signal
var fftx = fft(x) // perform FFT

def getReal(x: org.apache.commons.math3.complex.Complex) = x.getReal
var fftxReals = fftx map getReal
linePlotsOn
figure(1); subplot(2,1,1); plot(x); title("the signal");
subplot(2,1,2); plot(fftxReals);  title("FFT of the signal - real parts")
```

Matrix Select – Update Operations

It is convenient to be able to select/update Matrix ranges with a MATLAB-like style. ScalaLab (staring from Apr 17 version) improved on the routines that select and update Matrix sub-ranges. These routines have the same interface independent of the Matrix type. We illustrate those operations by means of examples.

Examples of Matrix range select operations

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

```
var N = 40; var M = 50;

var a = new Mat(N,M)

for (r<-0 until N)

  for (c<-0 until M)

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

We can select the row with index 2 with:

```
val ridx = 2

val ar2 = a(ridx, ::)
```

We can select rows 1 and 2 as:

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

or

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

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

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

Symmetrically we can work with columns:

```
val cidx = 3

val ac3 = a(::, cidx)

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

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

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

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

or

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

Examples of Matrix range update operations

Let create two simple zero-indexed matrices

var a = rand0(4, 8)

var o = ones0(2, 3)

The single element update operation by default does not resize the matrix, in order to avoid overhead. We can update a matrix range copying another matrix at the specified index as:

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

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

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

var al = rand0(15, 30)

var ol = ones0(12, 13)

var dx = 2; var dy = 3

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

Using a vector to fill Matrix rows or columns

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

val a = zeros0(10, 50)

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

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

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

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

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

```
// create a random matrix
var xx = rand(20)
var rowS = 1; var rowE = 16; var rowI = 2
var colS = 1; var colE = 16; var colI = 2
// select a row range
var ff = xx(rowS::rowE, ::)
// update that row range
xx(rowS::rowE, ::) = 8.8
// select a row range with increment rowI
var fff = xx(rowS::rowI::rowE, ::)
// update that row range
xx(rowS::rowI::rowE, ::) = 55.4

// select a column range
var ffc = xx(::, colS::colE)
// update that column range
xx(::, colS::colE)= -99.9
// select a column range with increment colI
var fffc = xx(::, colS::colI::colE)
// update that column range
xx(::, colS::colI::colE)= 33
// some more select operations
var sel1 = xx(rowS::2::rowE, 3::2::colE)+5.7
var sel2 = xx(rowS::rowE, 3::2::colE)+57.9
var sel3 = xx(rowS::rowE, 3::colE)
var sel4 = xx(rowS::rowE, 3::2::colE)
```

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

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

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

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

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

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

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

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

// And some update operations:

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

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

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

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

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

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

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

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

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

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

Consistency Across Many Matrix Libraries

An important advantage of ScalaLab is that it can utilize effectively and flexibly different

Java/Scala Matrix libraries. Each such library has its own data representations with the consequent advantages/disadvantages and implements its own numerical routines.

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

Many important classes of ScalaSci, such as the *Vec* , the one-indexed matrix class *Matrix* and the highly convenient *RichDouble1DArray* classes, are independent of the particular zero-indexed Matrix library. Therefore, these routines are always available, independently of the particular zero-indexed library at which the Scala Interpreter switches.

We placed the definitions of these libraries in a *Scala trait* , which all the library specific Scala objects mix-in, in order to obtain that functionality. This trait is the scalaSci.StaticScalaSciGlobal trait and can be obtained from the ScalaLab source.

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

Each scalaSci matrix wrapper class mixes-in the scalaSciMatrix trait, that can be obtained from the ScalaLab source code. We should note also that this trait defines a large number of **in-place** operations, e.g.

var a = ones0(20, 30)

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

Also, each ScalaSci Static Imports Object mixes in the StaticScalaSciCommonOps trait, in order to force the implementation of the common API. The current code of this object can also be obtained from the sources of ScalaLab.

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

The scalaSciMatrix trait

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

We describe some important operations provided by the *scalaSciMatrix* trait to any *scalaSci* matrix type.

Let create a 5X5 random matrix:

var x = rand0(5, 5)

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

var nr = x.numRows

var nc = x.numColumns

var nr2 = x.numRows()

var nc2 = x.numColumns()

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

var xv = x.getv

The *getNativeMatrixRef()* routine is also very useful. It returns *the library dependent class* that implements native operations. This allows ScalaLab code to combine Scala implemented operations, with the existing native operations provided by the Java library. In this way the full potential of the underlying Java class can be utilized. For example for an EJML based matrix type:

var nativeEJML = x.getNativeMatrixRef

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

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

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

Additionally, some very useful routines are defined as:

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

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

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

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

 /*returns the 2-norm condition number (the ratio of the
   largest singular value of X to the smallest). Large condition
   numbers indicate a nearly singular matrix.
 */
def cond(): Double
```

```
// provides an estimate of the number of linearly independent rows or columns of a matrix A.
def rank(): Int

/*   (V,D) = eig(X) : produces an array  D of eigenvalues and a
   full matrix V whose columns are the corresponding eigenvectors so
   that X*V = V*D */
 def eig(): (RichDouble2DArray, RichDouble1DArray)

/* (U,S,V) = svd() produces a diagonal matrix S, of the same
   dimension as X and with nonnegative diagonal elements in
   decreasing order, and unitary matrices U and V so that
   X = U*S*V'.
   */
  def  svd(): (RichDouble2DArray, RichDouble1DArray, RichDouble2DArray)

}
```

NUMAL library interface and the one-indexed Matrix class

Another basic library is the NUMAL Javal library described in [11]. NUMAL has a lot of routines covering a wide range of numerical analysis tasks. NUMAL uses one-indexing of arrays as MATLAB and Fortran also do. The *Matrix* ScalaLab class is a one-indexed class designed to facilitate the ScalaLab user in accessing NUMAL functionality. The routines of that library operate on the Array[Array[Double]] type and they are imported by default in the Scala Interpreter. Also, the *RichDouble2DArray* Scala object aims to implement an additional simpler interface to these

routines. Additionally, an implicit conversion from Matrix to Array[Array[Double]] allows the NUMAL machinery to be used with the Matrix ScalaLab type.

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

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

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

Constructors

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

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

Construction by specifying the initial elements, e.g.

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

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

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

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

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

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

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

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

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

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

Matrix(da: Array[Double])

var od = new Array[Double](10)

od(2)=2.2

var odm = new Matrix(od)

Basic Methods

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

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

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

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

Accessing the internal implemenation:

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

var ddArr = myMat.getv

Matrix number of Rows and can be retrieved as:

var Nrows = myMat.length

var Nrows = myMat.getv.length

var Ncols = myMat.getv(0).length

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

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

Resizing:

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

Convenient Matrix MATLAB-like operations:

a. Row selection

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Basic Matrix Routines

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

Aggregation Routines

- **Columnwise Sum, sum(a: Matrix): Array[Double]**
  returns the columnwise sums as an Array[Double], e.g.
  var a = ones1(4, 8)
  var ac = sum(a)

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

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

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

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

Routines

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

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

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

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

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

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

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

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

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

Description of the scalaSci.Mat class

The scalaSci.**Mat** class implements the default **zero-indexed** two-dimensional dense matrices in ScalaSci. We note that there are also some other **Mat** classes that implement zero-indexed matrices on-top of specific libraries, e.g. EJML, MTJ., Apache Common Maths etc.

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

Constructors

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

var m=new Mat(2, 3)

Construction by specifying the initial elements, e.g.

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

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

// construct Mat by copying the array values

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

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

dd(1)(1)=11

var mdd = new Mat(dd)

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

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

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

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

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

Mat(da: Array[Double])

var od = new Array[Double](10)

od(2)=2.2

var odm = new Mat(od)

Basic Methods

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

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

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

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

Routines

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

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

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

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

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

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

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

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

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