Tutorial on Matrix class

<u>Important tip</u>: Access elements of GroovyLab Matrices as Java double arrays, i.e.

x = rand(4,5)

x[2][3] = 23.23 // not x[2,3] = 23.23

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
```