**Announcement**

MDArray version 0.5.5 has Just been released. MDArray is a multi dimensional array implemented for JRuby inspired by NumPy ([www.numpy.org](http://www.numpy.org)) and Masahiro Tanaka´s Narray (narray.rubyforge.org). MDArray stands on the shoulders of Java-NetCDF and Parallel Colt. At this point MDArray has libraries for linear algebra, mathematical, trigonometric and descriptive statistics methods.

NetCDF-Java Library is a Java interface to [NetCDF files](http://www.unidata.ucar.edu/software/netcdf/index.html), as well as to many other types of scientific data formats.  It is developed and distributed by Unidata (<http://www.unidata.ucar.edu>).

Parallel Colt (<http://grepcode.com/snapshot/repo1.maven.org/maven2/net.sourceforge.parallelcolt/parallelcolt/0.10.0/>) is a [multithreaded](http://en.wikipedia.org/wiki/Thread_%28computer_science%29) version of [Colt](http://dsd.lbl.gov/~hoschek/colt/) (<http://acs.lbl.gov/software/colt/>). Colt provides a set of Open Source Libraries for High Performance Scientific and Technical Computing in Java. Scientific and technical computing is characterized by demanding problem sizes and a need for high performance at reasonably small memory footprint.

**What´s new:**

**Class MDMatrix and Linear Algebra Methods**

This version of MDArray introduces class MDMatrix. MDMatrix is a matrix class wrapping many linear algebra methods from Parallel Colt (see below). MDMatrix support only the following types: i) int; ii) long; iii) float and iv) double.

Differently from other libraries, in which matrix is a subclass of array, MDMatrix is a *twin* class of MDArray. MDMatrix is a *twin* class of MDArray as every time an MDMatrix is instantiated, an MDArray class is also instantiated. In reality, there is only one backing store that can be viewed by either MDMatrix or MDArray.

Creation of MDMatrix follows the same API as MDArray API. For instance, creating a double square matrix of size 5 x 5 can be done by:

matrix = MDMatrix.double([5, 5], [2.0, 0.0, 8.0, 6.0, 0.0,\

1.0, 6.0, 0.0, 1.0, 7.0,\

5.0, 0.0, 7.0, 4.0, 0.0,\

7.0, 0.0, 8.0, 5.0, 0.0,\

0.0, 10.0, 0.0, 0.0, 7.0])

Creating an int matrix filled with zero can be done by:

matrix = MDMatrix.int([4, 3])

MDMatrix also supports creation based on methods such as fromfunction, linspace, init\_with, arange, typed\_arange and ones:

* array = MDArray.typed\_arange("double", 0, 15)
* array = MDMatrix.fromfunction("double", [4, 4]) { |x, y| x + y }

An MDMatrix can also be created from an MDArray as follows:

* d2 = MDArray.typed\_arange("double", 0, 15)
* double\_matrix = MDMatrix.from\_mdarray(d2)

An MDMatrix can only be created from MDArrays of one, two or three dimensions. However, one can take a view from an MDArray to create an MDMatrix, as long as the view is at most three dimensional:

* # Instantiate an MDArray and shape it with 4 dimensions
* d1 = MDArray.typed\_arange("double", 0, 420)
* d1.reshape!([5, 4, 3, 7])
* # slice the array, getting a three dimensional array and from that, make a matrix
* matrix = MDMatrix.from\_mdarray(d1.slice(0, 0))
* # get a region from the array, with the first two dimensions of size 0, reduce it to a
* # size two array and then build a two dimensional matrix
* matrix = MDMatrix.from\_mdarray(d1.region(:spec => "0:0, 0:0, 0:2, 0:6").reduce)

printing the above two dimensional matrix gives us:

* matrix.print

3 x 7 matrix

0,00000 1,00000 2,00000 3,00000 4,00000 5,00000 6,00000

7,00000 8,00000 9,00000 10,0000 11,0000 12,0000 13,0000

14,0000 15,0000 16,0000 17,0000 18,0000 19,0000 20,0000

Every MDMatrix instance has a twin MDArray instance that uses the same backing store. This allows the user to treat the data as either a matrix or an array and use methods either from matrix or array. The above matrix can be printed as an array:

* matrix.mdarray.print

[[0.00 1.00 2.00 3.00 4.00 5.00 6.00]

[7.00 8.00 9.00 10.00 11.00 12.00 13.00]

[14.00 15.00 16.00 17.00 18.00 19.00 20.00]]

With an MDMatrix, many linear algebra methods can be easily calculated:

* # basic operations on matrix can be done, such as, ‘+’, ‘-‘, ´\*’, ‘/’
* # make a 4 x 4 matrix and fill it with ´double´ 2.5
* a = MDMatrix.double([4, 4])
* a.fill(2.5)
* # make a 4 x 4 matrix ´b´ from a given function (block)
* b = MDMatrix.fromfunction("double", [4, 4]) { |x, y| x + y }
* # add both matrices
* c = a + b
* # multiply by scalar
* c = a \* 2
* # divide two matrices. Matrix ´b´ has to be non-singular, otherwise an exception is
* # raised.
* # generate a non-singular matrix from a given matrix
* b.generate\_non\_singular!
* c = a / b

Linear algebra methods:

* # create a matrix with the given data
* pos = MDArray.double([3, 3], [4, 12, -16, 12, 37, -43, -16, -43, 98])
* matrix = MDMatrix.from\_mdarray(pos)
* # Cholesky decomposition from wikipedia example
* chol = matrix.chol
* assert\_equal(2, chol[0, 0])
* assert\_equal(6, chol[1, 0])
* assert\_equal(-8, chol[2, 0])
* assert\_equal(5, chol[2, 1])
* assert\_equal(3, chol[2, 2])

**MDArray and SciRuby:**

MDArray subscribes fully to the SciRuby Manifesto (<http://sciruby.com/>).

*“*[*Ruby*](http://www.ruby-lang.org/)*has for some time had no equivalent to the beautifully constructed [NumPy](http://numpy.scipy.org/), [SciPy](http://www.scipy.org/), and*[*matplotlib*](http://matplotlib.sourceforge.net/)*libraries for*[*Python*](http://www.python.org/)*.*

*We believe that the time for a Ruby science and visualization package has come. Sometimes when a solution of sugar and water becomes super-saturated, from it precipitates a pure, delicious, and diabetes-inducing crystal of sweetness, induced by no more than the tap of a finger. So is occurring now, we believe, with numeric and visualization libraries for Ruby.”*

**MDArray main properties are:**

* Homogeneous multidimensional array, a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers;
* Support for many linear algebra methods (see bellow);
* Easy calculation for large numerical multi dimensional arrays;
* Basic types are: boolean, byte, short, int, long, float, double, string, structure;
* Based on JRuby, which allows importing Java libraries;
* Operator: +,-,\*,/,%,\*\*, >, >=, etc.;
* Functions: abs, ceil, floor, truncate, is\_zero, square, cube, fourth;
* Binary Operators: &, |, ^, ~ (binary\_ones\_complement), <<, >>;
* Ruby Math functions: acos, acosh, asin, asinh, atan, atan2, atanh, cbrt, cos, erf, exp, gamma, hypot, ldexp, log, log10, log2, sin, sinh, sqrt, tan, tanh, neg;
* Boolean operations on boolean arrays: and, or, not;
* Fast descriptive statistics from Parallel Colt (complete list found bellow);
* Easy manipulation of arrays: reshape, reduce dimension, permute, section, slice, etc.;
* Reading of two dimensional arrays from CSV files (mainly for debugging and simple testing purposes);
* StatList: a list that can grow/shrink and that can compute Parallel Colt descriptive statistics;
* Experimental lazy evaluation (still slower than eager evaluation).

**Supported linear algebra methods:**

* backwardSolve: Solves the upper triangular system U\*x=b;
* chol: Constructs and returns the cholesky-decomposition of the given matrix.
* cond: Returns the condition of matrix A, which is the ratio of largest to smallest singular value.
* det: Returns the determinant of matrix A.
* eig: Constructs and returns the Eigenvalue-decomposition of the given matrix.
* forwardSolve: Solves the lower triangular system L\*x=b;
* inverse: Returns the inverse or pseudo-inverse of matrix A.
* kron: Computes the Kronecker product of two real matrices.
* lu: Constructs and returns the LU-decomposition of the given matrix.
* mult: Inner product of two vectors; Sum(x[i] \* y[i]).
* mult: Linear algebraic matrix-vector multiplication; z = A \* y.
* mult: Linear algebraic matrix-matrix multiplication; C = A x B.
* multOuter: Outer product of two vectors; Sets A[i,j] = x[i] \* y[j].
* norm1: Returns the one-norm of vector x, which is Sum(abs(x[i])).
* norm1: Returns the one-norm of matrix A, which is the maximum absolute column sum.
* norm2: Returns the two-norm (aka euclidean norm) of vector x; equivalent to Sqrt(mult(x,x)).
* norm2: Returns the two-norm of matrix A, which is the maximum singular value; obtained from SVD.
* normF: Returns the Frobenius norm of matrix A, which is Sqrt(Sum(A[i]2)).
* normF: Returns the Frobenius norm of matrix A, which is Sqrt(Sum(A[i,j]2)).
* normInfinity: Returns the infinity norm of vector x, which is Max(abs(x[i])).
* normInfinity: Returns the infinity norm of matrix A, which is the maximum absolute row sum.
* pow: Linear algebraic matrix power; B = Ak <==> B = A\*A\*...\*A.
* qr: Constructs and returns the QR-decomposition of the given matrix.
* rank: Returns the effective numerical rank of matrix A, obtained from Singular Value Decomposition.
* solve: Solves A\*x = b.
* solve: Solves A\*X = B.
* solveTranspose: Solves X\*A = B, which is also A'\*X' = B'.
* svd: Constructs and returns the SingularValue-decomposition of the given matrix.
* trace: Returns the sum of the diagonal elements of matrix A; Sum(A[i,i]).
* trapezoidalLower: Modifies the matrix to be a lower trapezoidal matrix.
* vectorNorm2: Returns the two-norm (aka euclidean norm) of vector X.vectorize();
* xmultOuter: Outer product of two vectors; Returns a matrix with A[i,j] = x[i] \* y[j].
* xpowSlow: Linear algebraic matrix power; B = Ak <==> B = A\*A\*...\*A.

**Properties´ methods tested on matrices:**

* density: Returns the matrix's fraction of non-zero cells; A.cardinality() / A.size().
* generate\_non\_singular: Modifies the given square matrix A such that it is diagonally dominant by row and column, hence non-singular, hence invertible.
* diagonal?: A matrix A is diagonal if A[i,j] == 0 whenever i != j.
* diagonally\_dominant\_by\_column?: A matrix A is diagonally dominant by column if the absolute value of each diagonal element is larger than the sum of the absolute values of the off-diagonal elements in the corresponding column.
* diagonally\_dominant\_by\_row?: A matrix A is diagonally dominant by row if the absolute value of each diagonal element is larger than the sum of the absolute values of the off-diagonal elements in the corresponding row.
* identity?: A matrix A is an identity matrix if A[i,i] == 1 and all other cells are zero.
* lower\_bidiagonal?: A matrix A is lower bidiagonal if A[i,j]==0 unless i==j || i==j+1.
* lower\_triangular?: A matrix A is lower triangular if A[i,j]==0 whenever i < j.
* nonnegative?: A matrix A is non-negative if A[i,j] >= 0 holds for all cells.
* orthogonal?: A square matrix A is orthogonal if A\*transpose(A) = I.
* positive?: A matrix A is positive if A[i,j] > 0 holds for all cells.
* singular?: A matrix A is singular if it has no inverse, that is, iff det(A)==0.
* skew\_symmetric?: A square matrix A is skew-symmetric if A = -transpose(A), that is A[i,j] == -A[j,i].
* square?: A matrix A is square if it has the same number of rows and columns.
* strictly\_lower\_triangular?: A matrix A is strictly lower triangular if A[i,j]==0 whenever i <= j.
* strictly\_triangular?: A matrix A is strictly triangular if it is triangular and its diagonal elements all equal 0.
* strictly\_upper\_triangular?: A matrix A is strictly upper triangular if A[i,j]==0 whenever i >= j.
* symmetric?: A matrix A is symmetric if A = tranpose(A), that is A[i,j] == A[j,i].
* triangular?: A matrix A is triangular iff it is either upper or lower triangular.
* tridiagonal?: A matrix A is tridiagonal if A[i,j]==0 whenever Math.abs(i-j) > 1.
* unit\_triangular?: A matrix A is unit triangular if it is triangular and its diagonal elements all equal 1.
* upper\_bidiagonal?: A matrix A is upper bidiagonal if A[i,j]==0 unless i==j || i==j-1.
* upper\_triangular?: A matrix A is upper triangular if A[i,j]==0 whenever i > j.
* zero?: A matrix A is zero if all its cells are zero.
* lower\_bandwidth: The lower bandwidth of a square matrix A is the maximum i-j for which A[i,j] is nonzero and i > j.
* semi\_bandwidth: Returns the semi-bandwidth of the given square matrix A.
* upper\_bandwidth: The upper bandwidth of a square matrix A is the maximum j-i for which A[i,j] is nonzero and j > i.

**Descriptive statistics methods imported from Parallel Colt:**

auto\_correlation, correlation, covariance, durbin\_watson, frequencies, geometric\_mean, harmonic\_mean, kurtosis, lag1, max, mean, mean\_deviation, median, min, moment, moment3, moment4, pooled\_mean, pooled\_variance, product, quantile, quantile\_inverse, rank\_interpolated, rms, sample\_covariance, sample\_kurtosis, sample\_kurtosis\_standard\_error, sample\_skew, sample\_skew\_standard\_error, sample\_standard\_deviation, sample\_variance, sample\_weighted\_variance, skew, split, standard\_deviation, standard\_error, sum, sum\_of\_inversions, sum\_of\_logarithms, sum\_of\_powers, sum\_of\_power\_deviations, sum\_of\_squares, sum\_of\_squared\_deviations, trimmed\_mean, variance, weighted\_mean, weighted\_rms, weighted\_sums, winsorized\_mean.

**Double and Float methods from Parallel Colt**:

acos, asin, atan, atan2, ceil, cos, exp, floor, greater, IEEEremainder, inv, less, lg, log, log2, rint, sin, sqrt, tan.

**Double, Float, Long and Int methods from Parallel Colt**:

abs, compare, div, divNeg, equals, isEqual (is\_equal), isGreater (is\_greater), isles (is\_less), max, min, minus, mod, mult, multNeg (mult\_neg), multSquare (mult\_square), neg, plus (add), plusAbs (plus\_abs), pow (power), sign, square.

**Long and Int methods from Parallel Colt**

and, dec, factorial, inc, not, or, shiftLeft (shift\_left), shiftRightSigned (shift\_right\_signed), shiftRightUnsigned (shift\_right\_unsigned), xor.

**MDArray installation and download:**

* Install Jruby
* jruby –S gem install mdarray

**MDArray Homepages:**

* <http://rubygems.org/gems/mdarray>
* <https://github.com/rbotafogo/mdarray/wiki>

**Contributors:**

Contributors are welcome.

**MDArray History:**

* 28/10/2013: Version 0.5.5 – class MDMatrix and Support for linear algebra;
* 07/08/2013: Version 0.5.4 – Support for NetCDF files reading and writing;
* 24/06/2013: Version 0.5.3 – Over 90% Performance improvements for methods imported from Parallel Colt and over 40% performance improvements for all other methods (implemented in Ruby);
* 16/05/2013: Version 0.5.0 - All loops transferred to Java with over 50% performance improvements. Descriptive statistics from Parallel Colt;
* 19/04/2013: Version 0.4.3 - Fixes a simple, but fatal bug in 0.4.2. No new features;
* 17/04/2013: Version 0.4.2 - Adds simple statistics and boolean operators;
* 05/04/2013: Version 0.4.0 – Initial release.