**Announcement**

MDArray version 0.5.3 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 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:**

**Performance Improvement**

On previous versions, array operations were done by passing a Ruby Proc to a loop for all elements of the given arrays. For instance, adding two MDArrays was done by passing Proc.new { |a, b| a + b } and looping through all elements of the arrays. Procs are very flexible in Ruby; however, from my experience with MDArray, also very slow.

On this version, when available, instead of passing a Proc to the loop, we pass a native Java method. Available Java methods are those extracted from Parallel Colt and listed below. Note that Parallel Colt has native methods for the following types only: “double”, “float”, “long” and “int”. With this change, there was a performance improvement of over 90%, and using MDArray operations is close to native Java operations. We expect (but have not yet benchmarking data) that this brings MDArray performance close to similar solutions such as NArray, NMatrix and NumPy (please try it, and if this assertion is false, I´ll be glad to change it in future announcements).

Methods not available in Parallel Colt but supported by Ruby, such as “sinh”, “tanh”, and “add” for “byte” type, etc. are still supported by MDArray. Again, to improve performance, instead of passing a Proc we now create a class as follows

class Add

def self.apply(a, b)

a + b

end

end

This change brought performance improvement of over 60% for MDArray operations with Ruby methods.

**Experimental Lazy Evaluation**

Usual MDArray operations are done eagerly, i.e., if @a, @b, @c are three MDArrays then the following:

@d = @a + @b + @c

will be evaluated as follows: first @a + @b is performed and stored in a temporary variable, then this temporary variable is added to @c. For large expressions, temporary variables can have significant performance impact.

This version of MDArray introduces lazy evaluation of expressions. Thus, when in lazy mode:

@lazy\_d = @a + @b + @c

will not evaluate immediately. Rather, the expression is preprocessed and only executed when required. Since at execution time the whole expression is known, there is no need for temporary variables as the whole expression is executed at once. To put MDArray in lazy mode we only need to set its mode to lazy with the following command “MDArray.lazy = true”. All expressions after that are by default lazy. In lazy mode, MDArray resembles Numexpr, however, there is no need to write the expression as a string, and there is no compilation involved.

MDArray does not implement broadcasting rules as NumPy. As a result, trying to operate on arrays of different shape raises an exception. On lazy mode, this exception is raise only at evaluation time, so it is possible to have an invalid lazy array. To evaluate a lazy array one should use the “[]” method as follows:

@d = lazy\_d[]

@d is now a normal MDArray.

Lazy MDArrays are really lazy, so let’s assume that @a = [1, 2, 3, 4] and @b = [5, 6, 7, 8]. Lets also have @l\_c = @a + @b. Now doing @c = @l\_c[], will evaluate @c to [6, 8, 10, 12]. Now, let´s do @a[1] = 20 and then @d = @l\_c[]. Now @d evaluates to [25, 8, 10, 12] as the new value of @a is used.

Lazy arrays can be evaluated inside expressions:

@l\_c = (@a + @b)[] + @c

In this example, @l\_c is a lazy array, but (@a + @b) is evaluated when the “[]” method is called and then added to @c. If now the value of @a or @b is changed, the evaluation of @l\_c will not be changed as in the previous example.

Finally, laziness is contagious. So, let´s assume that we have @l\_c as above, a lazy array and we do MDArray.lazy = false. From this point on in the code, operations will be done eagerly. Now doing: @e = @d + @l\_c, @e is a lazy array as its construction involves a lazy array. One should be careful when in eager mode mixing lazy and eager arrays:

@c = @l\_a + (@b + @c)

then, with parenthesis, first (@b + @c) is evaluated eagerly and then added lazily to @l\_a, giving a lazy array.

In this version, Lazy evaluation is around 40% **less** efficient in one machine I tested up to approximately the same performance in another equipment than eager evaluation when only native Java methods (Parallel Colt methods described below) are used in the expression. If expression involves any Ruby method, evaluation of lazy expressions becomes much slower that eager evaluation. In order to improve performance, I believe that compilation of expression will be necessary.

**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;
* 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).

**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:**

* 24/06/2013: Version 0.5.0 – 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.