**Announcement**

SciCom version 0.2.2 has been released. SciCom (Scientific Computing) for Ruby brings the power of R to the Ruby community. SciCom is based on Renjin, a JVM-based interpreter for the R language for statistical computing.

**R on the JVM**

Over the past two decades, the R language for statistical computing has emerged as the de facto standard for analysts, statisticians, and scientists. Today, a wide range of enterprises – from pharmaceuticals to insurance – depend on R for key business uses. Renjin is a new implementation of the R language and environment for the Java Virtual Machine (JVM), whose goal is to enable transparent analysis of big data sets and seamless integration with other enterprise systems such as databases and application servers.

Renjin is still under development, but it is already being used in production for a number of client projects, and supports most CRAN packages, including some with C/Fortran dependencies.

**SciCom and Renjin**

SciCom integrates with Renjin and allows the use of R inside a Ruby script. In a sense, SciCom is similar to other solutions such as RinRuby, Rpy2, PipeR, etc. However, since SciCom and Renjin both target the JVM there is no need to integrate both solutions and there is no need to send data between Ruby and R, as it all resides in the same JVM. Further, installation of SciCom does not require the installation of GNU R; Renjin is the interpreter and comes with SciCom. Finally, although SciCom provides a basic interface to Renjin similar to RinRuby, a much tighter integration is also possible (see examples below).

**SciCom with Standard R Interface**

SciCom allows R programmers to use R commands inside a Ruby script in a way similar to RinRuby by calling method eval and passing to it an R script:



Programmers can also use here docs to integrate an R script inside a Ruby script. The next example show a model for predicting baseball wins based on runs allowed and runs scored. The data comes from Baseball-Reference.com.



The output of the program above is:



**The SciCom “language”**

SciCom also allows for implementing R scripts in a “language” that is just like Ruby, so that the developer does not need to know that she is actually writing an R script. All R methods are accessible through an R namespace.

The next script is the same baseball model done in R above using SciCom ‘language’:



We show bellow an example of calculating the correlation matrix without using the build-in functions. First this is done in an R script and then using SciCom:

Now the same code using SciCom



As another example, here is a SciCom script to print the number of days for every month is 2005:



As can be seen from these examples, R methods can be accessed through the R namespace in SciCom, so, R method ‘seq’ is called in SciCom as ‘R.seq’. R methods that are applied on objects can be called in two ways, either using the R namespace as in ‘R.factor’ or directly on the object, as in this case we did ‘cmonth.factor’.

This last example shows how SciCom allows method chaining, which is not possible in an R script.

**What´s New**

This version integrates SciCom with MDArray. 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 (<https://sites.google.com/site/piotrwendykier/software/parallelcolt>) 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.

**Converting MDArray to R Array (same backing store)**

An MDArray can be converted to an R array by calling method ‘R.md’.

First, let´s create an MDArray of shape [4, 3]:



This is arr1 as printed from MDArray:



Now, converting this array to an R array and printing it:



The result is:



One very important aspect of this conversion is that both the MDArray and the R array use the same backing store, and thus, this conversion does not do any copying and has very low cost. However, WITH GREAT POWER COMES GREAT RESPONSABILITIES: since MDArray and the R array have the same backing store, a change in MDArray will also change the value of the R array.

r\_matrix also changes... arr1 and r\_matrix have the same backing store. Changing the content of an MDArray that points to the same backing store as an R vector should be done with care. Renjin assumes that the vector will never change and delays calculation of the vector to the latest possible time. In this case, since the value of the vector is changing, one can get unexpected behaviour. Use with care. We could prevent MDArray from being editable, however, we believe that allowing access to the backing store will have important implications for performance. If we have indication that this is not a good thing, then we will remove MDArrays ability to change the backing store of a Vector.

**SciCom main properties are**

* Allows access to R scripts from inside Ruby scripts;
* Allows for R scripts written in R by accessing method ‘R.eval’;
* Allows R scripts to be embedded inside here docs in Ruby;
* Creates a new ‘language’ that allows regular Ruby scripts to call R methods in such a way that programmers can be unaware of the fact that they are using R (although, of course, knowing R is of great benefit).;
* Integrates with MDArray allowing multi-dimensional arrays to be slice and cut and passed to an R script.

**SciCom installation and download**

* Install Jruby
* jruby –S gem install scicom

**SciCom Homepages**

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

**Contributors**

Contributors are welcome.

**How SciCom Works**

Almost everything is dynamically dispatched to Renjin.  The magic is basically done by Ruby method 'method\_missing'.  I´ll describe a bit how the whole thing works.

Let´s begin with a simple example:

> vec = R.c(1, 2.45, 3)

Ruby will try to find method 'c' on R class, since there is no such method, Ruby calls 'method\_missing' passing the argument list to it.  'method\_missing' calls 'parse' to parse the arguments.  In this case, every argument is an Numeric and 'parse' will basically convert the argument list to a string: "(1, 2.45, 3)".  'method\_missing' then makes the following call:

# R.eval(" <method\_name> + <arguments> "), which in this case is:

> R.eval("c(1, 2.45, 3)")  giving us the desired result.

Now let´s do:

> vec2 = R.c(vec, 4, 5)

Again, 'method\_missing' is called, which calls 'parse'.  The first argument to parse is now a Ruby::Vector and parse does the following:

# create a temporary variable, lets call it sc\_1234

# let vec.sexp be the actual java sexp for this vector, then:

> R.eval("sc\_1234 = vec.sexp")  # this stores the java vector in the sc\_1234 variable

# finally return all parameters as a string: "(sc\_1234, 4, 5)"

# Now let Renjin do the work:

> R.eval("c(sc\_1234, 4, 5)")

# return of the above is the vector [1, 2.5, 3, 4, 5] as expected

# finally, we remove the temporary variable sc\_1234, from Renjin name space in the hope that # this will allow gc to work.

Every statement in SciCom is converted to an evaluable string that can be passed to Renjin 'eval'.  Now, this adds some overhead, but unless we put a SciCom statment inside a large loop this should be reasonable.  One way that integration could be improved is if there was some way of calling 'eval' on a SEXP and not on a string.  Maybe you have this already, but I couldn´t find a way of doing it.

Now, if we have

> vec.mean

Then again we call 'method\_missing' of the Vector class.  In this case, first we check if 'mean' is a named element of the vector.  If it is, then we call:

> R.eval("vec[mean]")

If mean is not a named element of the vector, then we call:

R.eval("mean(vec)")

Which calculates the mean of the vector.  Again, this is just translating Ruby statements onto R evaluable strings.

Some new methods had to be defined.  For instance we have method '\_' in order to simulate methods such as %in%.  So here we do

> vec.\_ :in, vec2

This is a call to the '\_' method with 2 arguments ':in' and 'vec2'.  This translates to the correct R call:

> R.eval("vec %in% vec2")

In place of ':in' we can put any other string.

Methods and variable that have a '.' on their names are used in SciCom with '\_\_' notation.  So, read.csv in R is written as read\_\_csv.  'method\_missing' converts '\_\_' to '.' before calling eval.  So, if there is any variable in R written with '\_\_' there will be no way to access it, other than calling R.eval directly.

'method\_missing' is really the great magic!

**SciCom History:**

* 19/Nov/2014: Another small bug fix
* 16/Nov/2014: Version 0.2.1 – Small bug fix
* 16/Nov/2014: Version 0.2.0 – Initial release