Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

UNDER REVIEW

jvmr: Integration of R, Java, and Scala

David B. Dahl
Brigham Young University

Richard D. Payne
Texas A&M University

Deepthi Uppalapati
Capital One

Abstract

This paper covers the usage and implementation of the **jvmr** package for the statistical software R. The package provides four simple interfaces between R and programs or scripts running on the Java Virtual Machine. The package allows programmers to easily combine code from different languages in a single project. First, the package provides the means to embed an R interpreter in a Scala program using Scala-specific syntactical features. Second, more generally, the package allows a programmer to embed an R interpreter in a Java program or any other program written in a language available on the Java Virtual Machine (JVM), including JRuby, Jython, Groovy, Clojure. The **jvmr** package also provides the reverse functionality for embedding JVM languages in R using functionality from the **rJava** package. Specifically, the **jvmr** package embeds a Scala interpreter/compiler in R for running Scala code within an R program. Finally, it embeds the **BeanShell** interpreter in R, providing a high-level interface for running Java code snippets and calling Java code within an R program. The package is available on CRAN and requires no special installations or configurations of R, Java, or Scala. The **jvmr** package has been tested on a variety of operating systems, including Linux, Mac OS X, and Microsoft Windows.

Keywords: embedded interpreter, Java, Java Virtual Machine, R, Scala.

1. Introduction

The paper introduces the **jvmr** package that allow users to seamlessly incorporate R (R Core Team 2013) and Scala (Odersky and et. al. 2004) or Java code in one program and utilize each language's respective strengths (including libraries, methods, speed, graphics, etc.). Java is arguably one of the most popular programming languages currently in use. It is a general-purpose, object-oriented programming language which runs on the Java Virtual Machine (JVM). Scala is a newer, general purpose programming language that provides both object-oriented and functional programming constructs, freeing the programmer to intermingle paradigms. Scala also runs on the JVM, is statistically typed, and is often more concise and expressive than equivalent Java code. R is a scripting language and environment developed by statisticians for statistical computing and graphics with a large library of routines. R has many contributors and a large base of statistically-oriented users. Computationally intensive code written in R is generally slower than Scala, Java, C, Fortran, etc. Scala, Java, and R each have distinct strengths and weaknesses. Through the **jvmr** package, users can use the appropriate language or library *within the same script or program.*

The **jvmr** package provides a simple syntax to interface R, Scala, Java, and other JVM-based languages. Specifically, The **jvmr** package provides four high-level interfaces which embed R within Scala, R within Java, Scala within R, and Java within R. The user may "pull" and "push" values between languages. Presently, the **jvmr** package supports the pulling of single values, arrays, and matrices of boolean, integer, double, and string types.

Creating interfaces between programming languages is common. R has native ability to execute code written in C, C++, and Fortran. **rJava** (Urbanek 2013c) permits R to execute code written in Java and, more generally, any code running on the Java Virtual Machine (JVM). Conversely, the development community has also provided various means to run R code in other programming environments. These projects include **RServe** (Urbanek 2013b), **RJI** (Urbanek 2013a), **RPy** (Moreira and Warnes 2011), and **RinRuby** (Dahl and Crawford 2009). Another interesting project is Renjin (Bertram and et. al. 2013), a new JVM-based interpreter for the R language.

There are several items that make the **jvmr** package unique. First, the **jvmr** package is the first to explicitly support running R from within a Scala program, including taking advantage of Scala syntax. Second, while the **rJava** package allows one to call code from Java and other JVM-based languages, the **jvmr** package provides a Java interpreter and a Scala interpreter/compiler (using the **rJava** package). Finally, a key feature of the **jvmr** package is its simplicity and ease of installation while still providing reasonable performance. The package is available on the Comprehensive R Archive Network (CRAN) and uses standard installations of R, Scala, and Java. It has been tested on Linux, Mac OS X, and Microsoft Windows. No special compiling, installation, or configuration of R, Scala, Java, or the firewall is needed.

2. Installation

The **jvmr** package is available on the Comprehensive R Archive Network (CRAN) and can be installed through R's graphical interface or by executing the following R code:

```
R> install.packages("jvmr")
```

Note that the **jvmr** package only needs a standard installation of R. No special compilation flags or installation procedures are needed. It was developed and tested using R 3.0.1, but the package is expected to work on earlier versions. The Java interpreter from **BeanShell** (Niemeyer 2011) and the Scala interpreter/compiler are distributed as part of the **jvmr** package. These do *not* need to be installed separately in order to embed Java or Scala interpreters in R. To do the reverse (i.e., embed the R interpreter in a Java or Scala program), development kits for Java or Scala programming must be installed.

The **jvmr** package was developed and tested using Java 7 (a.k.a., Java 1.7) and Scala 2.10, but Java 6 is sufficient. The **jvmr** package was tested on Windows 7, Ubuntu 12.04, Mac OS X 10.8 (Mountain Lion). Windows XP and mainstream Linux distributions should be compatible.

3. Supported data types

The interfaces provided in the **jvmr** package allow a programmer to "pull" and "push" data between R, Scala, and Java. The names of these data types differ by language. For example, when an R `character` is pulled into Scala or Java, it will become a `String`. The term "pull" is somewhat of a misnomer since the **jvmr** package does not actually move objects across platforms. Rather, **jvmr** makes a copy of the data.

The **jvmr** package supports four data types (booleans, integers, doubles, and strings) in three types (primitives, vectors, and matrices). Matrices in R are represented in Java and Scala as row-major arrays of arrays. Vectors in R are arrays in Java and Scala. The term "primitive" is used loosely to denote single piece of data not contained in a vector or matrix. Table 1 outlines the equivalent representations of the four data types and the three data structures in the three languages.

	Primitives	Vectors	Matrices

R	`a <- TRUE`
	`b <- 1L`
	`c <- 1.0`
	`d <- "a"`
		`a <- c(TRUE,FALSE)`
		`b <- c(1L,2L,3L)`
		`c <- c(1.0,2.0,3.0)`
		`d <- c("a","b","c")`
			`a <- matrix(c(TRUE,FALSE),nrow=2)`
			`b <- matrix(c(1L,2L),nrow=2)`
			`c <- matrix(c(1.0,2.0),nrow=2)`
			`d <- matrix(c("a","b"),nrow=2)`

Scala	`val a = true`
	`val b = 1`
	`val c = 1.0`
	`val d = "a"`
		`val a = Array(true,false)`
		`val b = Array(1,2,3)`
		`val c = Array(1.0,2.0,3.0)`
		`val d = Array("a","b","c")`
			`val a = Array(Array(true),Array(false))`
			`val b = Array(Array(1),Array(2))`
			`val c = Array(Array(1.0),Array(2.0))`
			`val d = Array(Array("a"),Array("b"))`

Java	`boolean a = true;`
	`int b = 1;`
	`double c = 1.0;`
	`String d = "a";`
		`boolean[] a = {true,false};`
		`int[] b = {1,2,3};`
		`double[] c = {1.0,2.0,3.0};`
		`String[] d = {"a","b","c"};`
			`boolean[][] a = {{true},{false}};`
			`int[][] b = {{1},{2}};`
			`double[][] c = {{1.0},{2.0}};`
			`String[][] d = {{"a"},{"b"}};`

Table 1: Equivalent data types in R, Scala, and Java.

4. Usage

The examples provided in this paper are generally shown as if at an R, Scala, or **BeanShell**'s Java prompt. R code is designated with "`R>`", Scala code with "`scala>`", and Java code (as if at a **BeanShell** prompt) with "`bsh %`". Although the examples provided are presented as if at a command line, they can easily be incorporated into an R, Scala, or Java scripts and programs. Comprehensive documentation for the **jvmr** package is provided by means of R documentation, Scaladoc, and Javadoc. This documentation is available at `http://dahl.byu.edu/software/jvmr`.

4.1. Embedding R in Scala

In order for a Scala script or program to embed an R interpreter, the classpath must contain the JAR file whose location is given by the `.jvmr.jar` object in R. That is, at the R prompt, query the path of the JAR file on the local machine as follows:

```
R> library(jvmr)
R> .jvmr.jar
```

Let `JVMR_JAR` denote a shell variable containing the file path from the above R statement. In Windows, the value of the `JVMR_JAR` variable will be similar to:

```
C:/Program Files/R/R-3.0.1/library/jvmr/java/jvmr_2.10-1.0.0.jar
```

Suppose "MyProgram.scala" in the current directory contains the code for a Scala program using the **jvmr** package. To compile and run, for example on Linux and Mac OS X, execute the following at the operating system shell:

```
> JVMR_JAR=$(R --slave -e 'library(jvmr); cat(.jvmr.jar)')
> scalac -cp "$JVMR_JAR" MyProgram.scala     # Compile step
> scala  -cp ".:$JVMR_JAR" MyProgram         # Run step
```

Note that Windows users should use a semi-colon in lieu of a colon in the above code. Also, the manner in which the shell variable `JVMR_JAR` is set and accessed will be different. To use R in the interactive Scala read-eval-print loop (REPL), start Scala using the same code as above but exclude `MyProgram`.

Using R's statistical routines and graphics within Scala is simple and intuitive. The following examples are presented as if at the Scala prompt (designated by "`scala>`"), but the same code can be included in a Scala program. An R interpreter is instantiated using the `RInScala` companion object to the `RInScala` class in the package `org.ddahl.jvmr` as follows:

```
scala> import org.ddahl.jvmr.RInScala
scala> val R = RInScala()
```

In this case, the `R` object provides the connection through which R code and data can be evaluated, transferred, and accessed. Multiple instances of the R interpreter are allowed, and each interpreter maintains its own workspace and memory (i.e., each interpreter creates a separate connection to a new R session).

Once an instance of the R interpreter has been created, R expressions can be evaluated using the `eval` method. For example, consider:

```
scala> R.eval("words <- 'This String was made in R'")
```

The above code assigns a character string to the variable `words` in R through the interpreter `R`. Alternately, some may prefer the following shortcut:

```
scala> R> "words <- 'This String was made in R'"
```

The above statement is a shorthand version of the `eval` method and is easy to remember since it resembles an R prompt. The `eval` and `R>` methods can also evaluate multi-line code by using triple quotes:

```
scala> R> """n <- 100
             set.seed(24)
             draws <- rnorm(n)
             stdev <- sd(draws)
       """
```

At this point, the R session has four variables (`words`, `n`, `draws`, `stdev`) in its workspace. In order to "pull" a string representation of an R expression, use the `capture` method, as follows:

```
scala> println(R.capture("words"))
[1] "This String was made in R"
```

Again the **jvmr** package provides a shorthand:

```
scala> R> "words"
[1] "This String was made in R"
```

The `apply` function evaluates expressions in R similar to the `eval` function; however, the `apply` function also returns a Scala object of type `Any`. The `apply` function also has two equivalents. The following three statements are equivalent and return the value of `name` in R as a Scala object of type `Any`:

```
scala> R.apply("words")
scala> R("words'")
scala> R.words
```

If another instance of an R interpreter is created, it maintains its own workspace and memory. For instance, variables defined in the interpreter `R` would be inaccessible by the interpreter `AA`:

```
scala> val AA = RInScala()   // New R interpreter
scala> AA> "words"           // Produces an Error!
scala> R> "words"
[1] "This String was made in R"
```

The **jvmr** package allows the user to pull single values, vectors, and matrices from R into Scala and vice-versa. The `toPrimitive`, `toVector`, and `toMatrix` methods create Scala representations of R objects (i.e. it "pulls" objects from R into Scala). Each of these methods takes a type argument (`String`, `Double`, `Int`, or `Boolean`) for the Scala object. This is illustrated in the following examples:

```
scala> println("The variance is "+math.pow(R.toPrimitive[Double]("stdev"),2))
The variance is 0.899108595474283
```

```
scala> print(R.toPrimitive[Int]("as.integer(n)"))
100

scala> R.toVector[String]("c('Hello','World','!')")
res0: Array[String] = Array(Hello, World, !)

scala> R.toMatrix[Boolean]("matrix(c(TRUE,FALSE,TRUE,FALSE),nrow=2)")
res1: Array[Array[Boolean]] = Array(Array(true, true), Array(false, false))
```

Creating R representations of Scala objects (i.e. pulling Scala data into R) can be done with the `update` method, or its equivalent "dot" notation. For example, the following two lines of code are equivalent and assign the vector named "lengths" in R from a Scala array:

```
scala> R.update("lengths", Array[Double](2.2,3.5,4.2))
scala> R.lengths = Array[Double](2.2,3.5,4.2)
```

Although this "dot" syntax (second line of code above) is more concise, the `update` method will need to be used when assigning values to R variables containing a dot (e.g. `n.samples`). The **jvmr** package also provides the `prompt` method which opens an R prompt within Scala. To access the prompt, use either `R.prompt()` or `R>`. (Note: if invoking the R prompt using `R>`, the user may get a warning. This warning can be averted in three ways: 1. By simply invoking the prompt using `R.prompt()`, 2. Including `import scala.language.postfixOps` in the Scala program, or 3. Using "-language:postfixOps" as a command line option when invoking Scala.) Once the `prompt` method is invoked, the command line will act as if the user is at the R prompt. (On some operating systems, green text represents output from the R interpreter):

```
scala> R.prompt()
Welcome to the R prompt! Type R commands. No prompt is provided.
Exit by pressing ^D.
pval <- function(z) pnorm(z)
pval <- function(z) pnorm(z)
pval(0)
pval(0)
[1] 0.5
# Exit by pressing ^D (ctrl + D)
Exiting R prompt.
```

Variables and functions defined in the prompt will be available for use by the interpreter after the prompt is closed:

```
scala> R> "pval(0)"
[1] 0.5
```

4.2. Embedding R in Java

Embedding R in Java requires that additional JAR files are in the classpath. To find all the required JAR files, query the `.jvmr.alljars` object in R as follows:

```
R> library(jvmr)
R> .jvmr.alljars
```

The value of the `.jvmr.alljars` will be specific to the operating system and R installation. These JAR files should be included in classpath when compiling (e.g., using `javac`) and executing (e.g., using `java`).

Since **BeanShell** is packaged with the **jvmr** package, a Java read-eval-print loop (REPL), functionality from the **jvmr** package can be invoked from the operating system shell using:

```
> java -cp "$JVMR_ALL_JARS" bsh.Interpreter
```

assuming the `JVMR_ALL_JARS` shell variable contains the value of the `.jvmr.alljars` object in R.

Embedding R in Java in similar to embedding it in Scala, with a few syntactical limitations imposed by Java. In the following examples "`bsh %`" indicates Java code at the **BeanShell** prompt; however, the same code can be incorporated into a Java program. An R interpreter is instantiated using the `RInJava` class in the package `org.ddahl.jvmr` as follows:

```
bsh % import org.ddahl.jvmr.RInJava;
bsh % RInJava R = new RInJava();
```

The interpreter `R` provides the interface through which R code will be evaluated. As in Scala, multiple instances of the R interpreter are allowed and each interpreter will maintain its own workspace and memory.

Once an instance of the R interpreter has been created, evaluate R expressions within Java using the `eval` and `capture` methods in the same manner as in Scala. Note, however, that Java syntax does not support the "dot" notation, multi-line expressions, and the `R>` method. Processing multiple R statements within a single `eval` method is demonstrated in the following example:

```
bsh % R.eval("set.seed(24);\n" +
             "draws <- rgamma(100,1,2.3);\n" +
             "stdev <- sd(draws)");
```

As in Scala, the `apply` method evaluates R code and returns an object of class `Object` (Java's equivalent to Scala's `Any`). In Java, there is no shorthand for the `apply` method. The following code assigns the R variable `name` and returns a `JavaObject`:

```
bsh % R.apply("name <- 'Marilyn Monroe'");
```

To create Java copies of R objects (i.e. "pull" R objects into Java), the following methods are available:

```
toPrimitiveString   toPrimitiveDouble   toPrimitiveInt   toPrimitiveBoolean
   toVectorString      toVectorDouble      toVectorInt      toVectorBoolean
   toMatrixString      toMatrixDouble      toMatrixInt      toMatrixBoolean
```

Consider a couple examples:

```
bsh % double spread = R.toPrimitiveDouble("stdev");
bsh % double[] sample = R.toVectorDouble("draws");
```

Creating R copies of Java objects (i.e. "pulling" Java data into R) is the same as in Scala except Java code is used in place of Scala code. The following code creates an R matrix from a Java two-dimensional array:

```
bsh % R.update("rmatrix",new double[][] { {1.1,1.2}, {2.1,2.2}, {3.1,3.2} } );
bsh % System.out.print(R.capture("rmatrix"));
      [,1] [,2]
[1,]  1.1  1.2
[2,]  2.1  2.2
[3,]  3.1  3.2
```

Due to limitations on Java syntax, the "dot" syntax is not available in Java.

Similar to Scala, the `prompt` method opens an R prompt within Java. To access the prompt, type the command `R.prompt()`. Once the `prompt` method is invoked, the command line will act as if the user is at the R prompt (as shown in Section 4.1). As in Scala, functions and objects defined while in the R prompt will be available for use by the interpreter.

4.3. Embedding Scala in R

Instantiating a Scala interpreter/compiler in R is accomplished as follows:

```
R> library(jvmr)
R> a <- scalaInterpreter()
```

Multiple interpreters can be created and each maintains its own workspace and memory. Scala code can be evaluated using the `interpret` function or its shorthand equivalent. The following two lines of code are equivalent:

```
R> interpret(a,'val mu = 3')
R> a['val mu = 3']
```

Both the `interpret` function and its shorthand are capable of handling multi-line code:

```
R> a['val sigma = 2.5
      val n = 10
     ']
```

To create R representations of Scala objects (i.e. "pulling" Scala data into R) simply use `a['x']`, where `x` is a Scala object (or a valid Scala expression) and `a` is the interpreter created by the `scalaInterpreter` function. Copies of R objects can be created from Scala single values, arrays, and arrays of arrays of strings, doubles, integers, and booleans. Pulled values and objects can be assigned to variables in R or used elsewhere in the R code. Consider, for example:

```
R> r.mu <- a['mu']
R> r.sample <- rnorm(a['n'],a['mu'],a['sigma'])
R> r.mtx <- a['Array(Array(4.24,19.90),Array(1.3,5.2))']
R> r.mtx
     [,1] [,2]
[1,] 4.24 19.9
[2,] 1.30  5.2
```

Creating a Scala representation of an R variable or vector (i.e. "pulling" a value from R into Scala) can be done intuitively:

```
R> a['pVal'] <- pnorm(-2.5)
R> a['message'] <- c("Hello","world","again","!")
```

Output generated by Scala is not displayed in the R session by default, but can be enabled by setting the `echo.output` argument to `TRUE`. For example, consider:

```
R> a['message.foreach{ x => println("<" + x + ">")}']
R> a['message.foreach{ x => println("<" + x + ">")}',echo.output=TRUE]
<Hello>
<world>
<again>
<!>
```

String substitutions in Scala code is supported as additional arguments to the `interpret` function. Up to nine string substitutions are allowed and their placement is indicated in the string as `${d}`, where d is a digit `1`, `2`, ..., `9`. A simple example of substitution is:

```
R> a['val statement = "${1} costs ${2}."',"Milk","$3.70."]
statement: String = Milk costs $3.70.
```

Numerical arguments are converted to strings for substitution. The following example creates an array filled with 3 random doubles:

```
R> a['val draws = new Array[Double](${1})
      val random = new java.util.Random()
      for(i <- 0 until ${1}) draws(i) = random.nextDouble()', 3]
```

Note the use of multi-line code in the previous example.

4.4. Embedding Java in R

Embedding Java in R is identical to embedding Scala in R with two notable exceptions. First, the interpreter is created using the `javaInterpreter()` function (e.g. `a <- javaInterpreter()`). Second, the interpreter only processes Java code. It may be interesting to the user to note that when processing single Java statements, the semicolon may be omitted (e.g. `a['String name = "John"']`). Multi-line statements require the usual semicolons in standard Java code.

5. Case studies

5.1. Scala web server performs statistical analysis using R

Suppose a web developer in the Department of Natural Resources needs to build an online tool for field scientists studying the birth weights of a large mammal in separate herds. The data are entered into the web-based system running in the Play Framework available for Scala and is stored in an array of arrays of `Doubles` named `births`. Using the **jvmr** package, the developers can embed R in Scala to compute the p-value for a simple ANOVA comparing the average birth weights between herds. To accomplish this, the developer performs a standard installation of R on the web server, adds to Play's classpath the JAR whose location is given by the `.jvmr.jar` variable, and uses the following code:

```
scala> import org.ddahl.jvmr.RInScala
scala> val R = RInScala()
scala> R.update("births",births)
scala> R> """
            births <- as.data.frame(births)
            names(births) <- c("weight","herd")
            model <- with(births,lm(weight ~ as.factor(herd)))
            results <- anova(model)
          """
scala> val pValue = R.toPrimitive[Double]("results$'Pr(>F)'[1]")
```

The developer can then return the `pValue` variable in Scala as part of a larger web page resulting from an HTTP response. Of course, the Scala web server could also easily serve plots generated in R through the **jvmr** package.

5.2. Java program summarizes data using R

For simplicity, assume that the `birth` variable from the previous example is available in a Java program and that the developer wants the first quartile of the second herd. This can be accomplished with the following Java code:

```
bsh % import org.ddahl.jvmr.RInJava;
bsh % RInJava R = new RInJava();
bsh % R.update("births",births);
bsh % System.out.println("The first quartile is " +
          R.toPrimitiveDouble("quantile(births[births[,2]==2,1],0.25)"));

The first quartile is 9.795
```

5.3. Computationally-expensive algorithm compared in R, Scala, and C

Suppose one is interested in modeling the relationship between poverty and years of education via Bayesian logistic regression in R. For simplicity, consider only one covariate and ignore the intercept. The dependent variable is "poverty status" (with 1 indicating poverty and 0 indicating not in poverty) and our independent variable is "years of education." Assume a normal prior distribution with mean 1 and variance 1 for the coefficient on "years of education." Since the model is not conjugate, consider Markov chain Monte Carlo (MCMC) using a Metropolis sampling algorithm to obtain draws from the posterior distribution of the coefficient on "years of education." Due to the computationally intensive nature of MCMC methods, we demonstrate below that implementation using Scala is much faster than the pure R implementation and on par with an implementation using C. The Scala implementation is nonetheless very convenient because the code is embedded within the R script in a single file and a seperate compiler does not need to be installed.

Consider the side-by-side listing of code in Table 2 showing an implementation written purely in R and an implementation embedding Scala code in an R script. The implementation using Scala is a bit longer than the pure R implementation for two reasons: First, data must be passed between the two languages. Second, this Scala implementation uses an imperative programming style. An implementation in Scala using a declarative programming style would be both shorter and somewhat slower to execute. An implementation using C is also considered. Its code, available in Appendix A, follows the Scala code very closely except for syntactical differences between the languages.

Each implementation was run 10 times in a random order and the CPU time was recorded. The results are found in Table 3. Note that the Scala implementation is much faster than the pure R implementation and not much slower than the implementation using C. As opposed to the C implementation, an external compiler is not required for Scala implementation. A C compiler is not typically installed on Windows computers. It is also interesting to note that the Scala implementation is conveniently self-contained in a single file.

5.4. Exploiting a Java library in R

A feature of the **jvmr** package is the ability to easily use libraries within Java and Scala in R. By including the appropriate JAR files in the class path when instantiating the interpreter within R, software written in Scala and Java are available in R. This feature is particularly useful when the user is already familiar with a Java or Scala library and that equivalent functionality is not available in R.

Suppose a programmer is familiar with the `MixtureMultivariateNormalDistribution` class in the Apache Commons Math package to sample from a mixture of multivariate normal distributions. Certainly sampling from a mixture of multivariate normal distributions could be implemented by the programmer, but it is probably more convenient to use the Java library with which he is already familiar. To generate the samples, the user could either write an external Java program, use the **rJava** package in R, or use the **jvmr** package. Below we show an example using the **jvmr** package. Assume that the Apache Commons Math JAR file is in the current working directory and consider the following code and the resulting Figure 1.

```
library(jvmr)
a <- javaInterpreter("commons-math3-3.2.jar")

a['
import org.apache.commons.math3.distribution.*;
```

Implementation using pure R	Implementation using Scala embedded in R
`y <- c( 0,1,1, 0, 0, 0, 0, 0, 1, 1, 1)`	`y <- c( 0,1,1, 0, 0, 0, 0, 0, 1, 1, 1)`
`x <- c(16,6,9,18,20,19,13,14,11,10, 4)`	`x <- c(16,6,9,18,20,19,13,14,11,10, 4)`
	`library(jvmr)`
	`a <- scalaInterpreter()`
	`a['x'] <- x; a['y'] <- y`
	`a['`
	`import math.{exp,log}`
	`val random = new java.util.Random()`
`mcmc.in.R <- function(n.draws) {`	`def mcmc(nDraws : Int) = {`
`  log.like <- function(beta) {`	`  def logLike(beta : Double) = {`
`    sum(dbinom(y,1,1/(1+exp(-x*beta)),log=TRUE))`	`    var sum = 0.0`
`  }`	`    var i = 0`
	`    while ( i < y.length ) {`
	`      val p = 1/(1+exp(-x(i)*beta))`
	`      if ( y(i) == 0 ) sum += log(1-p)`
	`      else sum += log(p)`
	`      i += 1`
	`    }`
	`    sum`
	`  }`
`  samples <- rep(1,n.draws)`	`  val samples = Array.fill(nDraws) { 1.0 }`
`  ll.previous <- log.like(samples[1])`	`  var llPrevious = logLike(samples(0))`
	`  var i = 1`
`  for ( i in 2:length(samples) ) {`	`  while ( i < nDraws ) {`
`    proposal <- rnorm(1) + 1`	`    val proposal = random.nextGaussian() + 1`
`    ll.proposal <- log.like(proposal)`	`    val llProposal = logLike(proposal)`
`    hastings.ratio <- exp(ll.proposal - ll.previous)`	`    val hastingsRatio = exp(llProposal - llPrevious)`
`    if ( runif(1) < hastings.ratio ) {`	`    if ( random.nextDouble() < hastingsRatio ) {`
`      samples[i] <- proposal`	`      samples(i) = proposal`
`      ll.previous <- ll.proposal`	`      llPrevious = llProposal`
`    }`	`    }`
`    else samples[i] <- samples[i-1]`	`    else samples(i) = samples(i-1)`
`  }`	`    i += 1`
	`  }`
`  samples`	`  samples`
`}`	`}`
	`']`
	`mcmc.in.Scala <- function(ndraws) { a['mcmc(${1})',ndraws] }`

Table 2: A line-by-line comparison of two implementations of an algorithm in pure R and Scala embedded within R.

Implementation	Mean (St. Dev.) CPU Time in sec.	Times Faster Than R	Times Slower Than C
Pure R	175.2 (0.03)	1.0	22.0
Scala embedded in R	11.6 (0.12)	15.2	1.5
C called from R	7.9 (1.18)	22.2	1.0

Table 3: Execution speed for three implementations of a computationally-intensive algorithm.

```
double[] weights = {0.2,0.3,0.5};
double[][] means = {{0,0},{3,0},{6,0}};
double[][][] covariance = {{{2.0, 1.4},{ 1.4,2.0}},
                           {{1.0, 0.0},{ 0.0,1.0}},
                           {{1.0,-0.8},{-0.8,1.0}}};
MixtureMultivariateNormalDistribution multivar =
   new MixtureMultivariateNormalDistribution(weights,means,covariance);
']

n <- 1000000L
sample <- a['multivar.sample(${1})',n]
library(MASS)
density3d <- kde2d(sample[,1],sample[,2],n=60)
pdf("normal-mixture.pdf",width=7,height=6)
persp(density3d, box=FALSE, col="tomato",phi=5,theta=15)
dev.off()
```

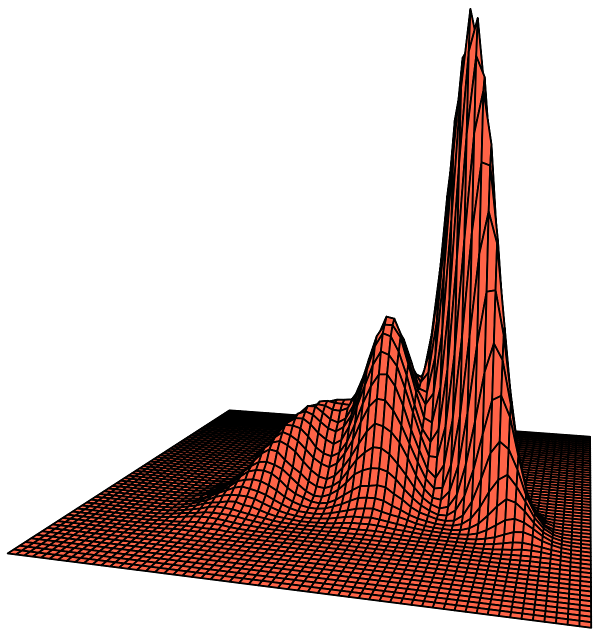

Figure 1: Multivariate density plot produced by R from data generated using a Java library.

6. Technical notes

In terms of the technical implementation, the embedding of R within Scala is accomplished by running a subprocess and utilizing a custom protocol over `standard input/output` and local TCP/IP sockets. The Scala software starts an R subprocess on the local machine, automatically finds available ports, and then sends commands to the R subprocess through `standard input/output` instructing the R client how to connect back to the Scala server. For security purposes, the server only accepts connections from the local host and automatically authenticates using a random challenge/response scheme. Commands are issued over the `standard input/output` and data is transferred over local TCP/IP sockets. Only the one Java Archive (JAR) file, whose location is given by the `.jvmr.jar` object in the **jvmr** package, is needed in the classpath.

The software embedding R within Java is accomplished using a Java class that simply wraps the Scala code just described. Since the Java code is essentially calling bytecode compiled from code written Scala, the Scala library and compiler JARs must also be the classpath, whose location is given by the `.jvmr.alljars` object in the **jvmr** package.

The embedding of Scala and Java withing R is based on executing functions in the **rJava** package which interface with **BeanShell**'s Java interpreter and Scala's interpreter/compiler. As these interpreters and the **rJava** package are already available from other authors, the amount of code and effort required to provide the embedding of these interpreters in R was considerably less than the effort required to do the opposite, i.e., provide the embedding of R in Java and Scala programs.

When repeatedly running the same Scala code embedded in R, it can be much faster to execute the loop in Scala as opposed to R. As an illustration, consider computing the standard normal distribution on a grid between −3 and 3 using loops in Scala and R.

```
library(jvmr)
a <- scalaInterpreter()
a['import math._; val pi = ${1}; val m = ${2}; val s = ${3}',pi,0,1]

looping <- function(n.items) {
  x <- seq(-3, 3, length.out=n.items)

  ##### Looping in Scala by calling Scala code once
  inScala <- system.time({
    a['x'] <- x
    evals <- a['x.map{x => exp(-(0.5*log(2*pi*s*s)+0.5*((x-m)/s)*((x-m)/s)))}']
  })

  ##### Looping in R by calling Scala code each time
  inR <- system.time({
    evals <- numeric(length(x))
    for ( i in 1:length(x) ) {
      a['x'] <- x[i]
      evals[i] <- a['exp(-(0.5*log(2*pi*s*s)+0.5*((x-m)/s)*((x-m)/s)))']
    }
  })
  cat(inR/inScala,"\n")
}
```

A call to `looping(n)` means there will be n function evaluations. When $n = 10$, R looping is 1.5 times slower than the equivalent Scala code. The problem becomes more apparent as n increases, with $n = 100$ and $n = 1000$ showing that R looping is 15 and 246 times slower, respectively. The explanation for these results are that each call to execute Scala code involves an on-the-fly compilation of the code to Java bytecode before execution on the JVM. Reducing the number of compilations avoids this costly step and allows the Java HotSpot compiler the opportunity to optimize the execution.

On a related note, it should be recognized that the **BeanShell** Java interpreter executes Java code by costly reflection. The code is not compiled and therefore not subject to Java HotSpot optimization. This is in contrast to the Scala interpreter/compiler which automatically compiles Scala code on-the-fly. Therefore, running Java code in the **BeanShell** interpreter is expected to be slow, whereas Scala code in the interpreter is expected to run at the full speed of the JVM. Of course, any compiled libraries executed by the **BeanShell** or Scala interpreters will themselves run at full speed on the JVM.

7. Conclusion

The **jvmr** package provides four interfaces which embed R in Scala, R in Java, Scala in R, and Java in R. The package has some limitations. The **jvmr** package is released under the GNU General Public License (GPL) version 3. Contributions are welcome. Currently, only booleans, integers, doubles, and strings in matrices, vectors, or single items are supported. It would be interesting to extend support to other data types or data structures. It would also be interesting to explicitly support other JVM-based languages.

Acknowledgements

David B. Dahl's work on this paper is supported by NIH NIGMS R01 GM104972.

References

Bertram A, et al (2013). "**Renjin**: JVM-based Interpreter for the R Language for Statistical Computing." URL https://code.google.com/p/renjin/.

Dahl DB, Crawford S (2009). "**RinRuby**: Accessing the R Interpreter from Pure Ruby." *Journal of Statistical Software*, **29**(4), 1–18. ISSN 1548-7660. URL http://www.jstatsoft.org/v29/i04.

Moreira W, Warnes GR (2011). "**RPy**: A Simple and Efficient Access to R from Python." URL http://rpy.sourceforge.net/.

Niemeyer P (2011). "**BeanShell**: Lightweight Scripting for Java." URL https://code.google.com/p/beanshell2/.

Odersky M, et al (2004). "An Overview of the Scala Programming Language." *Technical Report IC/2004/64*, EPFL, Lausanne, Switzerland.

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

Urbanek S (2013a). "**JRI** - Java/R Interface." URL http://www.rforge.net/JRI/.

Urbanek S (2013b). "**Rserve**: Binary R server." URL http://www.rforge.net/Rserve/.

Urbanek S (2013c). "**rJava**: Low-level R to Java Interface." URL http://www.rforge.net/rJava/.

Appendix A. C code for computationally-expensive algorithm

The R wrapping functions for the C code used in Section 5.3 is listed below:

```
y <- c( 0,1,1, 0, 0, 0, 0, 0, 1, 1, 1)
x <- c(16,6,9,18,20,19,13,14,11,10, 4)

dyn.load("C.so")
mcmc.in.C <- function(n.draws) {
  out <- .C("mcmcInC",
    nDraws=as.integer(n.draws),
    nObs=as.integer(length(x)),
    y=as.double(y),
    x=as.double(x),
    samples=double(n.draws))
  return(out$samples)
}
```

The actual C code used in Section 5.3 is listed below:

```
#include <math.h>
#include <R.h>
#include <Rmath.h>

double logLike(double beta, int nObs, double *y, double *x) {
  double sum = 0.0;
  int i = 0;
  while ( i < nObs ) {
    double p = 1/(1+exp(-x[i]*beta));
    if ( y[i] == 0 ) sum += log(1-p);
    else sum += log(p);
    i += 1;
  }
  return sum;
}

void mcmcInC(int *nDraws, int *nObs, double *y, double *x, double *samples) {
  GetRNGstate();
  samples[0] = 1.0;
  double llPrevious = logLike(samples[0],*nObs,y,x);
  int i = 1;
  while ( i < *nDraws ) {
    double proposal = rnorm(0.0,1.0) + 1;
    double llProposal = logLike(proposal,*nObs,y,x);
    double hastingsRatio = exp(llProposal - llPrevious);
    if ( runif(0.0,1.0) < hastingsRatio ) {
      samples[i] = proposal;
      llPrevious = llProposal;
    }
    else samples[i] = samples[i-1];
    i += 1;
  }
  PutRNGstate();
}
```

Affiliation:

David B. Dahl
Associate Professor
Department of Statistics
Brigham Young University
223 TMCB
Provo, UT 84602
E-mail: dahl@stat.byu.edu
URL: http://dahl.byu.edu

Richard D. Payne
Graduate Student
Department of Statistics
Texas A&M University
3143 TAMU College Station, TX 77843
E-mail: paynedrichard@gmail.com

Deepthi Uppalapati
Senior Business Analyst
Capital One
7933 Preston Road
Plano, TX 75024
E-mail: deepthiu@tamu.edu

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume VV, Issue II *Submitted:* 2013-06-21
MMMMMM YYYY *Accepted:* yyyy-mm-dd