Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

Under Review

Software Version: 2.5.0, Document ID: 2017-11-24

Integration of R and Scala Using rscala

David B. Dahl
Brigham Young University

Abstract

The **rscala** software is a simple, two-way bridge between R and Scala that allows users to leverage the unique strengths of both languages in a single project. Scala classes can be instantiated from R and Scala methods can be called. Arbitrary Scala code can be executed on-the-fly from within R, inline Scala functions can be defined, and callbacks to R are supported. R packages can be developed using Scala. Conversely, **rscala** also enables R code to be embedded within a Scala application. The **rscala** package is available on CRAN and has no dependencies beyond base R and the Scala standard library.

Keywords: Java Virtual Machine, JVM, language bridges, R, Scala.

1. Introduction

This paper introduces **rscala** (Dahl 2017c), software that provides a bridge between R (R Core Team 2017) and Scala (Odersky *et al.* 2004). The goal of **rscala** is to allow users to leverage the unique strengths of Scala and R in a single program. For example, R packages can implement computationally-intensive algorithms in Scala and, conversely, Scala applications can take advantage of the vast array of statistical packages in R. Callbacks from embedded Scala into R are supported. The **rscala** package is available on the Comprehensive R Archive Network (CRAN). Also, R can be embedded within a Scala application by adding a one-line dependency declaration in **Scala Build Tool** (SBT).

Scala is a general-purpose programming language that is designed to strike a balance be-

tween execution speed and programmer productivity. Scala programs run on the Java Virtual Machine (JVM) at speeds comparable to Java. Scala features object-oriented, functional, and imperative programming paradigms, affording developers flexibility in application design. Scala code can be concise, thanks in part to type inference, higher-order functions, multiple inheritance through traits, and a large collection of libraries. Scala also supports pattern matching, operator overloading, optional and named parameters, and string interpolation. Scala encourages immutable data types and pure functions (i.e., functions without side-effects) to simplify parallel processing and unit testing. In short, the Scala language implements many of the most productive ideas in modern computing. To learn more about Scala, we suggest *Programming in Scala* (Odersky, Spoon, and Venners 2016) as an excellent general reference.

Because Scala is flexible, concise, and quick to execute, it is emerging as an important tool for scientific computing. For example, **Spark** (Zaharia, Xin, Wendell, Das, Armbrust, Dave, Meng, Rosen, Venkataraman, Franklin, Ghodsi, Gonzalez, Shenker, and Stoica 2016) is a cluster-computing framework for massive datasets written in Scala. Several books have been published recently on using Scala for data science (Bugnion 2016), scientific computing (Jancauskas 2016), machine learning (Nicolas 2014), and probabilistic programming (Pfeffer 2016). We believe that Scala deserves consideration when looking for an efficient and convenient general-purpose programming language to complement R.

R is a scripting language and environment developed by statisticians for statistical computing and graphics. Like Scala, R supports a functional programming style and provides immutable data types. Scala programmers who learn R (and vice versa) will find many familiar concepts, despite the syntactical differences. R has a massive user base of statisticians and over 11,000 actively-maintained packages on CRAN. Hence, the Scala community has a lot to gain from an integration with R.

R code can be very concise and expressive, but may run significantly slower than compiled languages. In fact, computationally intensive algorithms in R are typically implemented in compiled languages such as C, C++, Fortran, and Java. The **rscala** package adds Scala to this list of high-performance languages that can be used to write R extensions. The **rscala** package is similar in concept to **Rcpp** (Eddelbuettel and François 2011), an R integration for C and C++, and **rJava** (Urbanek 2016), an R integration for Java. Though the **rscala** integration is not as comprehensive as **Rcpp** and **rJava**, it provides the following important features to blend R and Scala. First, **rscala** allows arbitrary Scala snippets to be included within an R script and Scala objects can be created and referenced directly within R code. These features allow users to integrate Scala solutions in an existing R workflow. Second, **rscala** supports callbacks to R from Scala, which allow developers to implement general, high-performance algorithms in Scala (e.g., root finding methods) based on user-supplied R functions. Third, **rscala** supports developing R packages based on Scala which allows Scala developers to make their work available to the R community. Finally, the **rscala** software makes it easy to incorporate R in a Scala application without even having to install the R package. In sum, **rscala**'s feature-set makes it easy to exploit the strengths of R and Scala in a single project.

We now discuss the implementation of **rscala** and some relevant existing work. Since Scala code compiles to Java byte code and runs on the JVM, one could access Scala from R via **rJava** and then benefit from the speed of shared memory. We originally implemented our **rscala** bridge using this technique, but later moved to a custom TCP/IP protocol for the following reasons. First, **rJava** and Scala's read-eval-print loop (REPL) are both implemented using custom class

loaders which, in our experience, conflict with each other in some cases. Second, since **rJava** links to a single instance of the JVM, one **rJava**-based package can configure the JVM in a manner that is not compatible with a second **rJava**-based package. The current **rscala** package creates a new instance of the JVM for each Scala instance to avoid such conflicts. Third, the simplicity of no dependencies beyond Scala's standard library and base R is appealing from a user's perspective. Finally, callbacks in **rJava** are provided by the optional JRI component, which is only available if R is built as a shared library. While this is the case on many platforms, it is not universal and therefore callbacks could not be a guaranteed feature of **rscala** software if it were based on **rJava**'s JRI.

The discussion of the design of **rscala** has so far focused on accessing Scala from R. The **rscala** software also supports accessing R from Scala using the same TCP/IP protocol. This ability is an offshoot of the callback functionality. Since Scala can call Java libraries, those who are interested in accessing R from Scala should also consider the Java libraries **Rserve** (Urbanek 2013) and **RCaller** (Satman 2014). **Rserve** is also "a TCP/IP server which allows other programs to use facilities of R" (`http://www.rforge.net/Rserve`). **Rserve** clients are available for many languages including Java. **Rserve** is fast and provides a much richer API than **rscala**. Like **rJava**, however, **Rserve** also requires that R be compiled as a shared library. Also, Windows has some limitations such that **Rserve** users are advised not to "use Windows unless you really have to" (`http://www.rforge.net/Rserve/doc.html`).

The paper is organized as follows. Section 2 describes using Scala from R. Some of the more important topics presented there include the data types supported by **rscala**, embedding Scala snippets in an R script, accessing precompiled Scala code from R, defining inline Scala functions, and calling back into R from embedded Scala. We also discuss how to develop R packages based on Scala. Section 3 describes using R from Scala. In both Sections 2 and 3, concise examples are provided to help describe the software's functionality. Section 4 provides a case study to show how Scala can easily be embedded in R to significantly reduce computation time for a simulation study. We conclude in Section 5 with future work.

2. Accessing Scala in R

This section provides a guide to accessing Scala from R. Those interested in the reverse — accessing R from Scala— will also benefit from understanding the ideas presented here.

2.1. Package and Scala installation

The **rscala** package is available on the Comprehensive R Archive Network (CRAN) and can be installed by executing the following R expression.

```
install.packages('rscala')
```

The **rscala** package requires a Scala installation in the 2.11.x or 2.12.x series. A convenience function, `rscala::scalaInstall()`, is provided to download and install Scala in the user's home directory under the `.rscala` directory. Because this is a user-level installation, administrator privileges are not required. System administrators can execute `rscala::scalaInstall(global=TRUE)`, which places Scala in the package's directory but requires a new installation every time the package is updated. To avoid this, system administrators can install Scala using their operating system's software management system (e.g.,

"`sudo apt install scala`" on Debian/Ubuntu based systems). Finally, both administrators and users can use a manual installation as described on the Scala webpage.

2.2. Instantiating a Scala interpreter in R

Load and attach the **rscala** package in an R session with the `library` function:

```
library('rscala')
```

Create a Scala instance using the `scala` function:

```
scala()
```

This implicitly makes the interpreter instance `s` available in the current environment. (The name can be customized with the `assign.name` option of the `scala` function.) Information on the Scala instance `s` is available using

```
scalaInfo(s)
```

Alternatively, details on the search for a suitable Scala installation are shown using

```
scalaInfo(verbose=TRUE)
```

The `scala` function includes parameters to specify which Scala installation to use, the class path, whether matrices are in row-major or column-major order, and several other settings. The functions `scala2` and `scala3` are equivalent to `scala` except they change the `mode` option to customize how the bridge between Scala and R is established. Interactive users may notice that the `scala` function feels faster because it starts Scala in the background. Details on this and all other functions are provided in the R documentation for the package (e.g., `help(scala)`).

A Scala session is only valid during the R session in which it is created and cannot be saved and restored through, for example, the `save` and `load` functions. Multiple Scala instances can be created in the same R session. Each Scala instance runs independently with its own memory and classpath.

The R to Scala bridge is not thread-safe so multiple R processes/threads should not access the same Scala instance simultaneously.

2.3. Calling Scala code from R

Evaluating Scala snippets

Snippets of Scala code can be compiled and executed within an R session using several operators. The most basic operator is `%@%` which evaluates Scala code and returns `NULL`. Consider, for example, computing the binomial coefficient $\binom{10}{3} = \prod_{i=1}^{3}(10-i+1)/i$. The code below uses Scala's `val` statement to define an immutable variable `c_10_3`. The expression `1 to 3` creates a range and the higher-order `map` method of the range applies the function `(10-i+1) / i.toDouble` to each element `i` in the range. Finally, the results are multiplied together by the `product` method.

```
s %@% '
  val c_10_3 = (1 to 3).map( i => {
    (10-i+1) / i.toDouble
  }).product.toInt
'
```

This result is available in subsequent Scala expressions as demonstrated below.

```
s %@% 'print("10 choose 3 is " + c_10_3 + ".")'

## 10 choose 3 is 120.
```

Notice the side effect of printing `120` to the console. The behavior for console printing is controlled by the arguments `serialize.output`, `stdout`, and `stderr` of the `scala` function. Default values depend on the operating system and are set such that console output is displayed in typical environments.

Scala snippets can also be evaluated with the `%~%` operator. Whereas `%@%` always returns `NULL`, `%~%` returns the result of the last expression in the Scala snippet.

```
tenChooseThree <- s %~% '(1 to 3).map( i => (10-i+1) / i.toDouble ).product'
tenChooseThree == choose(10,3)

## [1] TRUE
```

String interpolation

The **rscala** package features string interpolation for dynamic code snippets. R code placed between "`@{`" and "`}`" in a Scala snippet is evaluated and replaced by the string representation of the R expression's value before the Scala snippet is executed. The R code is executed in the same environment (i.e., scope) as the evaluation request. A snippet can contain any number of `@{...}` expressions. For example,

```
n <- 10
k <- 3
label <- "number of threesomes among ten people"
s %@% '
  val count = (1 to @{k}).foldLeft(1) { (prod,i) => prod * (@{n}-i+1)/i }
  println("The @{label} is " + count + ".")
'

## The number of threesomes among ten people is 120.
```

Care is needed when using string interpolation because it relies on R's character representation on an R expression. One might be surprised, for example, that the second expression in the next example is false. This is because `@{tenChooseThree^20}` is replaced by `6.191736e+20` which, when parsed by the Scala compiler, leads to a slightly different value than the calculated value.

```
s %~% 'math.pow(count, 20) == @{tenChooseThree^20}'

## [1] FALSE
```

Primitive and copyable types

A Scala result of class `Byte`, `Int`, `Double`, `Boolean`, or `String` is passed back to R as a length-one vector of raw, integer, double, logical, or character, respectively. We refer to these as the *primitive types* supported by the **rscala** package. Further, Scala arrays and rectangular arrays of arrays of the primitive types are passed to R as vectors and matrices of the equivalent R types. We call *copyable types* those types that are primitives, arrays of primitives, and rectangular arrays of arrays of the primitive types. The name emphasizes the fact that these data structures are serialized and copied between Scala and R. This may be a costly exercise for large data structures.

The code below produces a 2x5 matrix in R. If the `row.major` argument of the `scala` function is changed to `FALSE` when defining the Scala instance `s`, the code produces a 5x2 matrix instead.

```
s %~% 'Array.fill(2)(Array.fill(5)(scala.util.Random.nextDouble))'

##           [,1]      [,2]       [,3]       [,4]      [,5]
## [1,] 0.8307614 0.3680790 0.32608991 0.09047411 0.1829110
## [2,] 0.7239708 0.9187835 0.08826573 0.11337335 0.5833289
```

Table 1 shows the mapping of primitive Scala and R types using code examples.

Scala references

If the result of a Scala expression is not a copyable type, then the `%~%` operator will return a reference to a Scala object that can be used in subsequent evaluations. If a Scala reference is desired, even when working with copyable types, use the `%.~%` operator.

In the next example, an instance of the class `scala.util.Random` is created and, because the result is not a copyable type, a Scala reference is returned. Second, a Scala reference to an array of integers is returned, despite the fact this is a copyable type, because the `%.~%` operator is used.

```
rng <- s %~% 'new scala.util.Random()'
rng

## ScalaInterpreterReference... res6: scala.util.Random
## scala.util.Random@4b37b01e

oneToTenReference <- s %.~% 'Array.range(1,11)'
oneToTenReference

## ScalaInterpreterReference... res8: Array[Int]
## [I@4a52178
```

Primitives	Vectors / Arrays	Matrices / Rectangular Arrays of Arrays
`a <- as.raw(3)` `val a = 3.toByte`	`f <- as.raw(c(1, 2))` `val f = Array(1.toByte, 2.toByte)`	`k <- matrix(as.raw(c(1, 2)), nrow=2)` `val k = Array(Array(1.toByte), Array(2.toByte))`
`b <- TRUE` `val b = true`	`g <- c(TRUE, FALSE)` `val g = Array(true, false)`	`l <- matrix(c(TRUE, FALSE), nrow=2)` `val l = Array(Array(true), Array(false))`
`c <- 1L` `val c = 1`	`h <- c(1L, 2L, 3L)` `val h = Array(1, 2, 3)`	`m <- matrix(c(1L, 2L), nrow=2)` `val m = Array(Array(1), Array(2))`
`d <- 1.0` `val d = 1.0`	`i <- c(1.0, 2.0, 3.0)` `val i = Array(1.0, 2.0, 3.0)`	`n <- matrix(c(1.0, 2.0), nrow=2)` `val n = Array(Array(1.0), Array(2.0))`
`e <- "a"` `val e = "a"`	`j <- c("a", "b", "c")` `val j = Array("a", "b", "c")`	`o <- matrix(c("a", "b"), nrow=2)` `val o = Array(Array("a"), Array("b"))`

Table 1: Scala values of type `Byte`, `Int`, `Double`, `Boolean`, or `String` (labeled *primitives*), as well as arrays and rectangular arrays of arrays of these types are copied from Scala to R as length-one vectors, vectors, and matrices of the equivalent R types. These are called *copyable types*. Each cell in the table contains two lines: an R expression (top) and the equivalent Scala expression (bottom) with the same identifier. The matrix examples assume that the `scala` function is called with `row.major=TRUE`.

Getting and setting variables

Values of copyable types and Scala references can be obtained as the result of evaluating a Scala expression using the `%~%` and `%.~%` operators. The expression can be very complex or, as in the examples below, merely the name of a Scala identifier.

```
s %@% 'val fibSeq = Array[Double](0, 1, 1, 2, 3, 5, 8, 13, 21)'
fibSeqAsDouble  <- s %~%  'fibSeq'
fibSeqReference <- s %.~% 'fibSeq'
```

Scala values can also be obtained using the `$` operator and the identifier name. For example, the following both provide equivalent definitions for `fibSeqAsDouble` above.

```
fibSeqAsDouble <- s$fibSeq
fibSeqAsDouble <- s$val('fibSeq')
```

Likewise, an equivalent definition for `fibSeqReference` is

```
fibSeqReference <- s$.val('fibSeq')
```

Note that `val` is a reserved word in Scala. Therefore, using `val` and `.val` does not conflict with any variable names in Scala. While somewhat more verbose, the argument to `val` and `.val` can be a literal or a variable, whereas the `$` operator requires a literal.

Values of copyable types and Scala references can be set in the Scala session using assignment with the `$` operator, e.g.:

```
s$fibSeq <- c(0, 1, 1, 2, 3, 5, 8, 13, 21)
s$copyOfFibSeqReference <- fibSeqReference
```

Instantiating objects

Scala objects can be instantiated in three ways. The following example demonstrates functionally equivalent ways of creating a new instance of `scala.util.Random` with the seed set at 123.

```
seed <- 123L
rng <- s %.~% 'new scala.util.Random(@{seed})'
rng <- s$.scala.util.Random$new(seed)
rng <- s$do('scala.util.Random')$new(seed)
```

Each method differs in terms of flexibility, readability, and speed. The first is mostly Scala code and therefore self-evident to a Scala developer, but it is the slowest and awkward when arguments are not easily set with string interpolation. The second (using `s$.`) is concise, fast, and flexible in the arguments. The string literal following `s$.` and before the second `$` (e.g., `scala.util.Random`) can be the name of any class in the classpath. (The period in `s$.` is used to differentiate class names from variables names.) The last method (using `s$do`) is also fast and flexible but is slightly awkward. It has the added advantage, however, that the class name given as the argument to `s$do` can be a literal (as in this example) or a variable. Note that `do` is a reserved word in Scala and is therefore guaranteed not to conflict with a variable name.

Accessing methods and variables of Scala objects

Taking inspiration from **rJava**'s high-level `$` operator, methods associated with Scala references can be called directly using the `$` operator, as shown below.

```
rng$setSeed(24234L)
rng$nextInt(10L)

## [1] 4

oneToTenReference$sum()

## [1] 55
```

If the result of a method call on a Scala reference is not a copyable type, then a reference to a Scala object is returned. If a Scala reference is desired even when working with copyable types, add a final argument `.AS.REFERENCE=TRUE`. For example,

```
intReference <- rng$nextInt(10L, .AS.REFERENCE=TRUE)
```

The value of an instance variable may be accessed as if there was a method of the same name taking no arguments. For example, the value `self` in an instance of `scala.util.Random` is access as

```
rng$self()

## ScalaCachedReference... _: java.util.Random
## java.util.Random@2042a26f
```

If there are no arguments when calling a method of a Scala reference, the empty parentheses are excluded in the generated Scala code. These empty parentheses are needed, however, when one intends to use the methods default arguments. In that class, use `.PARENTHESES=TRUE` when calling the method.

Calling methods of singleton objects

In contrast to Java, Scala classes do not have static variables or methods. Equivalent functionality is provided by singleton objects in Scala. A companion object is a singleton object whose name is the same as a class. Methods of singleton objects can be called in three ways. For example, consider the companion object `Array` to the class `Array`. Its `range` method creates an array of regularly-spaced elements. The following three statements are all functionally equivalent:

```
oneToTenReference <- s %.~% 'Array.range(1, 11)'
oneToTenReference <- s$.Array$range(1L, 11L, .AS.REFERENCE=TRUE)
oneToTenReference <- s$do('Array')$range(1L, 11L, .AS.REFERENCE=TRUE)
```

As for instantiating objects, each approach has its advantages in terms of flexibility, readability, and speed.

Method arguments, null references, and length-one vectors

Arguments to a method of an object (as well as argument to `new`) can be copyable types and Scala references. To pass a null reference of a particular type, use the `scalaNull` function. For example, Java's `java.lang.System` has a static `setProperties` method which takes a null reference to `java.util.Properties` to clear the system properties, e.g.:

```
s$.java.lang.System$setProperties(scalaNull('java.util.Properties'))
```

R has no scalar types but they are often used in Scala. As such, length-one vectors have special semantics. In the R expression `rng$nextInt(10L)`, the value `10L` is an integer vector of length one in R, but is passed to Scala as `Int`, not `Array[Int]`. This is the most natural and convenient behavior. If, however, an R vector should always be passed as an array — despite the fact that it might be of length one — wrap the vector in a call to the `I` function. This ensures that the vector is treated "as is". For example, consider a singleton object with an `apply` method that takes an array of any arbitrary type `T` and a value of type `T`, and sets every element of the array to that value:

```
setter <- s %.~% '
  object setter {
    def apply[T](x: Array[T], value: T) = x.indices.foreach { x(_) = value }
  }
```

```
  setter
'
```

When calling the `apply` method of the `setter` object, the first argument must be an array. Thus, if there is a potential that the R vector is length-one, it should be wrapped by the `I` function. In the example below, the first argument is wrapped by `I` and is therefore passed to Scala as an array. The second argument is a length-one double vector in R yet is treated as a `Double` (instead of `Array[Double]`) because it is *not* wrapped by `I`:

```
arr <- s %.~% 'Array(math.Pi, math.E)'
arr$mkString("<", ", ", ">")

## [1] "<3.141592653589793, 2.718281828459045>"

setter$apply(I(arr), 3)
arr$mkString("<", ", ", ">")

## [1] "<3.0, 3.0>"
```

The `apply` *and* `update` *methods*

Scala users are aware of the "compiler magic" that injects calls to the `apply` method of objects when no method is specified. In the **rscala** package, this works for Scala snippets, but the `apply` method must be specified explicitly when using the `$` operator. For example, consider an array of the starting elements of the Fibonacci sequence and the following functionally-equivalent expressions:

```
fibSeqAsInt <- s %~% 'Array(0, 1, 1, 2, 3, 5, 8, 13, 21)'
fibSeqAsInt <- s %~% 'Array.apply(0, 1, 1, 2, 3, 5, 8, 13, 21)'
fibSeqAsInt <- s$.Array$apply(0L, 1L, 1L, 2L, 3L, 5L, 8L, 13L, 21L)
fibSeqAsInt <- s$do('Array')$apply(0L, 1L, 1L, 2L, 3L, 5L, 8L, 13L, 21L)
```

Likewise, the `update` method is automatically injected by the Scala compiler when appropriate, but must be explicit when using the `$` operator. Consider, for example, assigning the value of π to the second element of the array using the following. The last four statements are functionally equivalent.

```
s %@% 'val fibSeq = Array[Double](0, 1, 1, 2, 3, 5, 8, 13, 21)'
fibSeqReference <- s %.~% 'fibSeq'
s %@% 'fibSeq(1) = math.Pi'
s$.fibSeq$update(1L, pi)
s$do('fibSeq')$update(1L, pi)
fibSeqReference$update(1L, pi)
```

The previous example also illustrates that the `s$.` and `s$do` notations — which were first introduced for object instantiation and calling methods of singleton objects — can also be used for existing Scala values (e.g., `fibSeq` above).

Quoting method names

Scala has type parameterization which is similar but arguably more advanced than generics in Java and templates in C++. In many instances, the Scala compiler infers the type parameter, but the user may need or want to explicitly provide it. When using the `$` operator, the method name with its type parameter should be quoted to prevent parsing errors in R. The following expressions are functionally equivalent.

```
fibSeqAsDouble <- s %~% 'Array[Double](0, 1, 1, 2, 3, 5, 8, 13, 21)'
fibSeqAsDouble <- s %~% 'Array.apply[Double](0, 1, 1, 2, 3, 5, 8, 13, 21)'
fibSeqAsDouble <- s$.Array$'apply[Double]'(0L, 1L, 1L, 2L, 3L, 5L, 8L, 13L, 21L)
fibSeqAsDouble <- s$do('Array')$'apply[Double]'(0L, 1L, 1L, 2L, 3L, 5L, 8L, 13L, 21L)
```

Note the quotes around `apply[Double]` used in the last two expressions. Of course, since R treats numeric literals as doubles, the simplest way to get the same result from the `apply` method of the `Array` companion object is

```
fibSeqAsDouble <- s$.Array$apply(0, 1, 1, 2, 3, 5, 8, 13, 21)
```

Likewise, names of Scala methods may not be valid identifiers in R and may also need to be quoted to avoid parsing errors in R. For example, note that the method `:+` is quoted here:

```
list <- s$.List$apply(1L, 2L, 3L)
augmentedList <- list$':+'(100L)
paste0(augmentedList$toString(), " now contains 100.")

## [1] "List(1, 2, 3, 100) now contains 100."
```

2.4. Defining inline Scala functions

In addition to calling previously-compiled Scala methods, the **rscala** package enables Scala functions to be defined *within* an R session. The associated Scala code is compiled on-the-fly and cached for subsequent evaluation in the R session. This feature is inspired by the R packages **Rcpp** and **inline** (Sklyar, Murdoch, Smith, Eddelbuettel, Francois, and Soetaert 2015) for C, C++, and Fortran. We do not recommend that long, complicated algorithms be implemented as inline Scala functions, but these functions can be helpful for implementing small tasks or writing dynamic code to interface with existing Scala code.

To demonstrate Scala functions, consider computing the number of partitions of n items (i.e., the Bell number (Bell 1938) of n). This number is often used for finite mixture models and random partition models (e.g., Casella, Moreno, and Girón (2014)). First, consider this R implementation of an efficient algorithm based on the Bell triangle (also known as the Aitken's array or the Peirce triangle):

```
bell.version1 <- function(n, format=c("character","integer","double","log")[3]) {
  n <- as.integer(n[1])
  if ( n <= 0 ) stop("'n' must be at least 1.")
  if ( n == 1 ) return(1)
```

```
  r1 <- r2 <- numeric(n)
  r1[1] <- 1
  for ( k in 2:n ) {
    r2[1] <- r1[k-1]
    for ( i in 2:k ) r2[i] <- r1[i-1] + r2[i-1]
    r1 <- r2
  }
  value <- r2[n]
  if ( format == "character" ) sprintf("%0.0f", value)
  else if ( format == "integer" ) as.integer(value)
  else if ( format == "double" ) value
  else if ( format == "log" ) log(value)
}
```

The `bell.version1` function performs calculations based on double-precision floating-point arithmetic and provides *exact* answers for $n \leq 22$, gives *approximate* answers for $22 < n \leq 218$, and overflows for $n > 218$. The `format` argument controls the function's output. Unfortunately, $n \leq 218$ is quite limiting because sample sizes are often much larger in practice. The following Scala function, defined within the R session, implements the same algorithm but allows n to be greater than 218.

```
bell.version2 <- function(n, format=c("character","integer","double","log")[3]) {
  n <- as.integer(n[1])
  if ( n <= 0 ) stop("'n' must be at least 1.")
  s %!% '
    var r1 = new Array[BigInt](n)
    var r2 = new Array[BigInt](n)
    r1(0) = BigInt(1)
    for ( k <- 1 until n ) {
      r2(0) = r1(k-1)
      for ( i <- 1 to k ) r2(i) = r1(i-1) + r2(i-1)
      val tmp = r1; r1 = r2; r2 = tmp
    }
    val value = r1(n-1)
    format match {
      case "character" => value.toString
      case "integer" => value.toInt
      case "double" => value.toDouble
      case "log" =>
        val blex = value.bitLength - 1022
        if ( blex > 0 ) math.log( (value >> blex).toDouble ) + blex * math.log(2)
        else math.log(value.toDouble)
    }
  '
}
```

There are a few minor differences between the R and Scala versions (e.g., Scala uses zero-based indexing of arrays, syntactic differences between R and Scala, and the Scala version avoids the copying found in the R version), but the practical difference is that `bell.version2` uses infinite precision integer arithmetic based on Scala's builtin `BigInt` class. Overflow still occurs when transferring to R using `format="integer"` or `format="double"`, but there is

no overflow for almost any n when `format="log"`, and the exact value is returned when `format="character"`. The sample code below shows that the Scala function produces exact integer calculations for large n. (The timings saved in `cpuFirstEval` will be used later in Section 2.7.)

```
cpuFirstEval <- system.time(
   bigNumber <- bell.version2(500, format="character")
)
cat(paste(strsplit(bigNumber, "(?<=.{80})", perl = TRUE)[[1]], collapse="\n"), "\n")

## 160607260103999145374373286046550778629192454664500124922145864706366090316923887
## 422645330683773815475260839567013749555010376206442659914858239975604239194722 53
## 673154287711452243482625943425324523587807683222016163482602127637462832110631 75
## 471588305900499988767247495691030562568738615933054947931678912360815254163437 38
## 220590486226851969446456073386720128563218641743914456222275592531169402467217 92
## 836722819030355122406510335693235060922934260516725520015677030684701624259205 47
## 283590019445340218904698540925474839209070475849159426162420912714791185467698 39
## 437339847349415603411338019508936411672293535432986991680881631979332663613611 71
## 749567806252210577980695569672801349293276671456649407517188002835703107649119 16
## 148945975987515393829548296018963500962354552857468009581242277738079057682593 18
## 238628583897096173869307416513453942294577 72
```

Take a closer look at the definition of the `bell.version2` function. We call this a Scala function because it contains `s %!% '...'`, where `s` is a Scala instance and `'...'` represents a Scala snippet. The effect of the `%!%` operator is two-fold: i. it automatically makes the arguments of the enclosing function (namely, `n` and `format`) available in the Scala snippet, although this can be customized with the `[` operator (as explained below), and ii. it caches the on-the-fly compilation of the Scala snippet to substantially improve speed for repeated calls. The `%!%` operator returns a copyable type or, if the result is not a copyable type, a Scala reference. The `%.!%` operator is identical except that it always returns a Scala reference.

The consumer of a Scala function does not need to know that the implementation is written in Scala, but the programmer of a Scala function should bear in mind a few items. If an argument to a Scala function is not a copyable type or a Scala references, it will become a `EphemeralReference` object that can still be used in R code executed by the Scala function. (This functionality is demonstrated later.) The `scalaNull` function can be used to pass a null reference of a particular type. For example, an argument to a Scala function might be `rng = scalaNull("scala.util.Random")`, indicating a null reference to `scala.util.Random`.

Additional variables in the local environment can be automatically serialized to Scala by listing them after the `[` operator of the Scala instance. Conversely, if an argument is not needed in the Scala implementation (because, for example, it has already been processed in the R code), the serialization can be avoided by using the `drop` argument of the `[` operator. So, for example, we can make available new variables `x`, `y`, and `z` and skip variables `a` and `b` using `s["x","y","z",drop=c("a","b")] %!% '...'`. The automatically generated Scala code for the conversion of variables can be displayed (for debugging purposes) by setting the `show.snippet` option, e.g., `scalaSettings(s, show.snippet=TRUE)`.

2.5. Callbacks into R from embedded Scala

When a Scala instance is created with the `scala` function, an instance of the Scala class

`org.ddahl.rscala.RClient` is bound to the identifier `R`. This object provides access to the R session from within the Scala instance. The `RClient` class is thread-safe. Its source code and Scaladoc are located on GitHub: https://github.com/dbdahl/rscala/.

To assign a value to a variable in the R session from Scala, use the `set` method:

```
s %@% '
  R.set("zone", java.util.TimeZone.getDefault.getDisplayName)
  R.set("atLeast8", scala.util.Properties.isJavaAtLeast("1.8"))
'
zone

## [1] "Mountain Standard Time"

atLeast8

## [1] TRUE
```

R variables can be accessed in Scala using several methods. The first method is `get`. It returns a `Tuple2` where the first member is the R variable's value (statically typed as `Any`) and the second is a `String` identifying the resulting Scala type. Consider the example below in which the value of the R variable `T` is obtained. Although the runtime type of `T` is `Boolean`, the static type is `Any`. To be useful, it will likely need to be cast to another type as demonstrated with the call to `asInstanceOf[Boolean]` below.

```
s %@% '
  val result = R.get("T")
  println("The result is " + result)
  if ( result._1.asInstanceOf[Boolean] ) {
    println("Good, nobody messed with the value of T.")
  }
'


## The result is (true,Boolean)
## Good, nobody messed with the value of T.
```

Instead of casting the result from the `get` method, it may be more convenient to call a method that returns a specific type. The `RClient` class includes a suite of methods whose names start with `get` and end in XY, where $X \in \{$`R`, `I`, `D`, `L`, `S`$\}$ and $Y \in \{$`0`, `1`, `2`$\}$. The value of X indicates whether the result from R should be interpreted as raw, integer, double, logical, or character, respectively. The value of Y indicates whether the result should be interpreted as a scalar, an array, or a rectangular array of arrays, respectively. This example uses the `getL0` method to return the value of the variable `T` as a logical scalar value.

```
s %@% 'if ( R.getL0("T") ) { println("Good, nobody messed with the value of T.") } '

## Good, nobody messed with the value of T.
```

R expressions can be evaluated with a suite of methods whose names start with `eval` and end in XY, where X and Y have the same meaning as in the `get` methods. In the example below,

R and Scala are used together to sample from a chi-square distribution with 100 degrees of freedom.

```
set.seed(324)
s %~% 'R.evalD1("rnorm(100, sd=3)").map(math.pow(_, 2)).sum'

## [1] 866.7211
```

The `RClient` class also provides the ability to call R functions through methods that start with `invoke` and end in XY (as in the `get` and `eval` methods). For example, the previous example could also be implemented as follows.

```
set.seed(324)
s %~% '
  val mean = 100
  R.invokeD1("rnorm", mean, "sd" -> 3).map(math.pow(_, 2)).sum
'

## [1] 866.7211
```

The example above demonstrates the use of Scala's builtin notation for creating pairs (e.g., `"sd" -> 3`) to provide named arguments to the R function invoked from Scala. The arguments to an `invoke` method can be copyable types, Scala references, and `EphemeralReference` objects, the last being automatically generated in Scala functions for all arguments that are wrapped by the `II` function or that are not copyable types and not Scala references. Note that a `EphemeralReference` object is only valid within the Scala function in which it is defined. It can, however, be made valid beyond the Scala function by converting it to a `PersistentReference` using `R.getReference(x)`, where `x` is a `EphemeralReference` object.

A more interesting use case is calling a user-supplied R function from Scala. First, consider an R function that computes $f(n, \alpha)$, the expectation of the Ewens(n, α) distribution, i.e., the expected number of clusters when sampling n observations from a discrete random measure obtained from the Dirichlet process with mass parameter α.

```
f <- function(n,alpha) sapply(alpha, function(a) sum(a / (1:n + a - 1)))
f(100, 1.0)

## [1] 5.187378
```

In a Bayesian analysis, the Ewens distribution is a prior distribution in random partition models and α is a hyperparameter. In the prior elicitation process, practitioners may want to find the value of α that corresponds to the expert's anticipated number of clusters. Thus, the task is to numerically solve $f(n, \alpha) = \mu$ for α, given fixed values for n and μ. To be specific, suppose $n = 1000$ and $\mu = 10$. The value α can be obtained using root finding methods. Here, we demonstrate the bisection method implemented as a Scala function. The function's first argument, `func`, takes a user-defined R function. Since this argument is not a copyable type or a Scala reference, it is passed to Scala as an `EphemeralReference`, which is subsequently used in the expression `R.invokeD0(func, x)` to call the R function.

```
bisection <- function(func, lower=1.0, upper=1.0, epsilon=0.000000001) s %!% '
  def g(x: Double) = R.invokeD0(func, x)
  val (fLower, fUpper) = (g(lower), g(upper))
  if ( fLower * fUpper > 0 ) sys.error("lower and upper do not straddle the root.")
  @scala.annotation.tailrec
  def engine(l: Double, u: Double, fLower: Double, fUpper: Double): Double = {
    if ( math.abs( l - u ) <= epsilon ) ( l + u ) / 2
    else {
      val c = ( l + u ) / 2
      val fCenter = g(c)
      if ( fLower * fCenter < 0 ) engine(l, c, fLower, fCenter)
      else engine(c, u, fCenter, fUpper)
    }
  }
  engine(lower, upper, fLower, fUpper)
'

bisection(function(a) f(100, a) - 10, 0.1, 20)

## [1] 2.572197
```

The most important aspect of this example is found in the first line of the Scala function where the `invokeD0` method calls the `R` function referenced by `func` and returns the result as a `Double`.

The **rscala** package supports infinite recursion (subject to available resources) between R and Scala. For example, the `recursive.sum` function below repeatedly calls itself from Scala to compute $0 + 1 + 2 + \ldots + n$.

```
recursive.sum <- function(n=0L) s %!% '
  if ( n <= 0 ) 0 else n + R.invokeI0("recursive.sum", n - 1)
'
recursive.sum(10)

## [1] 55
```

2.6. Memory management

The **rscala** package ties into the garbage collectors of both R and Scala. As such, the user often does not need to think about memory management. There are a few important things to note, however. First, the default maximum heap size set by the Java Virtual Machine may not be sufficient. Adjust the heap size using the `scala` function's `heap.size` argument, or use the global option `rscala.heap.maximum` (e.g., `options(rscala.heap.maximum="4G")`). The former takes precedence over the latter.

Second, there is an unresolved issue (SI-4331) with the Scala REPL (read-eval-print-loop) where allocated memory cannot be freed even if the same identifier is set to another value. This issue prevents memory from being recovered in **rscala** when using the `%~%` and `%.~%` operators, or when using the the `$` assignment operator. This issue does not affect Scala functions and calls to methods on Scala references. Hence, we encourage developers to use

functions and methods for memory intensive applications. As will be shown later, functions and methods also enjoy faster execution than the equivalent code using the `%~%` and `%.~%` operators.

2.7. Speed considerations

Section 4 considers the easy of implementing and the execution speed of a simulation study in R, C++ via **Rcpp**, and Scala via **rscala**. It is not a comprehensive comparison of the performance of these languages. For that, we refer readers to benchmarks available on the web, including Gouy (2017). Here we wish to highlight performance characteristics of **rscala** itself.

Every Scala snippet associated with the `%@%`, `%~%`, and `%.~%` operators is compiled at every invocation. In contrast, a Scala snippet in a Scala function (see Section 2.4) and methods of Scala references are only compiled the first time they are run. Subsequent invocations are faster because the compiled code is cached and re-used. Recall that in Section 2.4 the variable `cpuFirstEval` saved the system time associated with the first invocation of the Scala function `bell.version2`.

```
cpuFirstEval

##    user  system elapsed
##   0.000   0.000   0.296
```

When we run the function again, we find the system time is substantially reduced because the code does not need to be compiled.

```
cpuSecondEval <- system.time(
   bigNumber <- bell.version2(500, format="character")
)
cpuSecondEval

##    user  system elapsed
##   0.004   0.000   0.036

cpuFirstEval['elapsed'] / cpuSecondEval['elapsed']

##  elapsed
## 8.222222
```

Methods of Scala references also benefit from caching. Consider, for example, two calls to the method `nextGaussian` of an instance of `scala.util.Random`.

```
rng <- s$.scala.util.Random$new()
first  <- system.time( rng$nextGaussian() )['elapsed']
second <- system.time( rng$nextGaussian() )['elapsed']
c(first=first, second=second, ratio=first/second)

##  first.elapsed second.elapsed  ratio.elapsed
##          0.111          0.001        111.000
```

Expression	Package	Q1	Mean	Median	Q3
fasterNextGaussianRJava()	rJava	31.00	39.14	39.27	43.20
fasterNextGaussian()	rscala	269.79	335.96	289.80	322.11
rngRJava$nextGaussian()	rJava	778.09	900.78	798.19	817.48
rng$nextGaussian()	rscala	424.37	470.37	448.36	477.58

Table 2: Comparison of execution time of various ways to call the `nextGaussian` method of an instance of the `scala.util.Random` class. Since the method itself is relatively fast, the timings here are an indication of the overhead involved with the various techniques. Each expression was evaluated 1000 times and the results are in microseconds.

Beyond the one-time cost of compiling, calling methods of Scala references still involves a recurring invocation cost, some of which can be eliminated as follows. Call the desired method of the Scala reference with an additional trailing argument `.EVALUATE=FALSE` and store the resulting function, e.g.:

```
fasterNextGaussian <- rng$nextGaussian(.EVALUATE=FALSE)
```

The function `fasterNextGaussian` is optimized and has less overhead than explicitly calling the `nextGaussian` method of a Scala reference. By way of comparison, **rJava** also provides two means to call the `nextGaussian` method. Suppose that `rngRJava` is the result of instantiating an object of class `scala.util.Random` using **rJava**. The high-level `$` operator of **rJava** can call this method using `rngRJava$nextGaussian()`. Alternatively, the **rJava**'s low-level interface provides the `.jcall` function. The next example and Table 2 compare the speed of **rscala**'s `rng$nextGaussian()` and its optimized `fasterNextGaussian()`, together with **rJava**'s two ways of calling the same method.

```
library('rJava', verbose=FALSE, quietly=TRUE)
invisible(
  rJava::.jinit(
    list.files(file.path(scalaInfo(s)$home, "lib"), full.names=TRUE)
  )
)
rngRJava <- rJava::.jnew("scala.util.Random")
fasterNextGaussianRJava <- function() rJava::.jcall(rngRJava, "D", "nextGaussian")

if ( suppressWarnings(require('microbenchmark', quietly=TRUE)) ) {
  timings <- summary(microbenchmark(fasterNextGaussianRJava(), fasterNextGaussian(),
    rngRJava$nextGaussian(), rng$nextGaussian(), times=1000))
} else load('timings.RData')
```

The results in Table 2 indicate that **rJava**'s `.jcall` interface is much faster than the other techniques. We recommend that **rscala** users avoid calling Scala code in long-running, tight inner loops where millisecond delays can add up.

2.8. Developing packages based on rscala

The **rscala** package enables developers to use Scala in their own R packages to implement computationally intensive algorithms. For example, the **shallot** (Dahl 2017b) and **bamboo**

(Dahl 2017a) packages on CRAN use Scala via **rscala** to implement statistical methodology of their associated journal articles (Dahl, Day, and Tsai 2017; Li, Dahl, Vannucci, Joo, and Tsai 2014). The **shallot** package takes advantage of **rscala**'s callback functionality to allow access a user-specific likelihood and sampling function. Readers are invited to study those examples in addition to our description here.

An R package based on **rscala** should include `rscala` in the `Imports` field of the package's `DESCRIPTION` file. Also, add `import(rscala)` to the `NAMESPACE` file. Define an `.onLoad` function which calls `.rscalaPackage(pkgname)`, where `pkgname` is the package's name. The `onLoad` function may be as simple as

```
.onLoad <- function(libname, pkgname) {
  .rscalaPackage(pkgname)
}
```

The `.rscalaPackage` function binds a Scala instance to the identifier `s` in the package's namespace using the `scala` function. The default for the `mode` argument is `"parallel"`. This instantiates a Scala interpreter in the background and therefore has minimal delay when the interpreter `s` is first used. If, however, Scala itself is not already installed, the user is asked for permission to download and install Scala (using the `scalaInstall` function). Analogous to **rJava**'s `.jpackage` function, the `.rscalaPackage` function adds the JAR files in the source package's `inst/java` directory to Scala's classpath. Since Scala is binary-compatible only within major releases, package developers are encouraged to cross-compile for releases 2.11 and 2.12 and to place compatible Scala JAR files in `inst/java/scala-2.11` and `inst/java/scala-2.12`, respectively. If a package supports only specific major releases of Scala, change the `major.release` argument of the `.rscalaPackage` function.

The `.rscalaPackage` function takes several optional arguments. The `classpath.packages` argument allows the package to use the JAR files of another package. For example, the **shallot** package uses this argument to import the Apache Commons Mathematics Library JAR files distributed with the R package **commonsMath** (The Apache Software Foundation 2017). The rational is that JAR files can be large and, by having them in a separate package, they do not need to be updated as frequently. The arguments `classpath.prepend` and `classpath.append` provide fine-grained control over the classpath. The argument `snippet` provides Scala expressions that will be evaluated when Scala is instantiated. This feature is useful for definitions and import statements. Finally, other arguments to `.rscalaPackage` are passed directly to the `scala` function. This `.onLoad` function taken from the **shallot** package demonstrates several of these optional arguments and also shows a callback to R so that random data generation in Scala is based on R's random number seed.

```
.onLoad <- function(libname, pkgname) {
  snippet <- '
    import org.apache.commons.math3.random.{ RandomDataGenerator => RDG }
    import org.ddahl.shallot._
    import parameter._
    import parameter.decay._
    import parameter.partition._
    import distribution._
    import mcmc._
```

```
    def rdg() = {
      val ints = R.evalI1("runif(2,-.Machine$integer.max,.Machine$integer.max)")
      val seed = ((ints(0).asInstanceOf[Long]) << 32) | (ints(1) & 0xffffffffL)
      val r = new RDG()
      r.reSeed(seed)
      r
    }

    // This circumvents a bug in the class loader on some versions of Scala/JVM.
    scala.util.Try {
      new org.apache.commons.math3.random.EmpiricalDistribution()
    }
  '
  ## Users may want to use 'options(rscala.heap.maximum="2G")'.
  .rscalaPackage(pkgname,classpath.packages="commonsMath",snippet=snippet)
}
```

A package's embedded Scala instance should be terminated when the package is unloaded by calling the `.rscalaPackageUnload` function in the `.onUnload` hook as shown here.

```
.onUnload <- function(libpath) {
  .rscalaPackageUnload()
}
```

Because **rscala**'s syntax for calling precompiled code is very similar to **rJava**'s high-level `$` operator, developing a package based on **rscala** can be very familiar to those accustomed to **rJava**. Take, for example, CRAN's **mailR** package: "Interface to Apache Commons Email to send emails from R" (Premraj 2015). This package uses **rJava** and its high-level `$` convenience operator. As a proof of concept, we ported the **mailR** package to **rscala**, replacing the dependency on **rJava**. Version 0.6 of the **mailR** package is available on GitHub at https://github.com/rpremraj/mailR/ and our port is at https://github.com/dbdahl/mailR/. The port involved changes to the `DESCRIPTION` file, the `NAMESPACE` file, and two script files. We deleted 16 lines, added 4 lines, and modified 15 lines. Most modifications were simple changes. For example,

```
base_dir <- .jnew("java.io.File", normalizePath(getwd()))
```

became

```
base_dir <- s$.java.io.File$new(normalizePath(getwd()))
```

The difference between the two versions can be viewed here: https://github.com/dbdahl/mailR/commit/feb911f. Of course, porting a package that makes frequent use of **rJava**'s "low-level interface" (e.g., the `.jcall` function) will require more changes.

The **bamboo** package was originally implemented in **rJava**. The original **rJava** code is commented out with equivalent **rscala** code following immediately after. See, for example, https://github.com/dbdahl/bamboo/blob/master/R/bamboo.R. This example also illustrates the difficulty in calling Scala using **rJava** since Scala has several features that do not map directly

to Java equivalents. Often, the developer is required to write Java-friendly wrapper methods in Scala that hide advanced Scala features.

3. Accessing R in Scala

So far we have demonstrated assessing Scala from R. Conversely, **rscala** can also embed an R interpreter in a Scala application via the `org.ddahl.rscala.RClient` class. This is achieved by generalizing the previously-discussed callback functionality. In this case, however, there is not an existing instance of the R interpreter. The R client spawns an R instance, immediately starts the embedded R server, and connects R to Scala.

The `RClient` class is thread-safe. Source code and Scaladoc are located on GitHub: https://github.com/dbdahl/rscala/. As a convenience, **rscala**'s JAR file is available in standard repositories for use by dependency management systems. To use `RClient` in a Scala application, simply add the following line to SBT's `build.sbt` file.

```
libraryDependencies += "org.ddahl" %% "rscala" % "2.5.0"
```

Note that, since the necessary R code is bundled in the JAR file, the **rscala** package does *not* need to be installed in R. An embedded R interpreter is instantiated as follows:

```
val R = org.ddahl.rscala.RClient()
```

This assumes that the registry keys option was not disabled during the R installation on Windows. On other operating systems, `R` is assumed to be in the search path. If these assumptions are not met or a particular installation of R is desired, the path to the R executable may be specified explicitly (e.g., `org.ddahl.rscala.RClient("/path/to/R/bin/R")`). By default, console output from R is serialized back to Scala. The protocol overhead may be reduced by using `serializeOutput=false` when instantiating an `RClient`.

The **rscala** package can be an easy and convenient way to access statistical functions, facilitate calculations, manage data, and produce plots in a Scala application. Consider, for example, wrapping R's `qnorm` function to define a method in Scala by the same name.

```
val R = org.ddahl.rscala.RClient()

def qnorm(x: Double, mean: Double = 0, sd: Double = 1, lowerTail: Boolean = true) = {
  R.invokeD0("qnorm", x, mean, sd, "lower.tail" -> lowerTail)
}

val alpha = 0.05
println(s"If Z is N(0,1), P(Z >= ${qnorm(alpha, lowerTail=false)}) = $alpha.")

// If Z is N(0,1), P(Z >= 1.6448536269514726) = 0.05.
```

The next example uses R's dataset `eurodist` to compute the European city that is closest, on average, to all other European cities. While this statistical calculation is easily implemented in R, one can imagine a Scala application that needs to perform a more taxing calculation that leverages R's rich data-processing functions.

```
val R = org.ddahl.rscala.RClient()
val distances = R.evalD2("as.matrix(eurodist)")
val cities = R.evalS1("attr(eurodist,'Labels')")
val centralCity = distances.map(_.sum).zip(cities).minBy(_._1)._2
println(s"Europe's central city is $centralCity.")

// Europe's central city is Lyons.
```

The `RClient` also enables a Scala application to access R's extensive plotting facilities and to take advantage of the many packages available in R. As an example of a nontrivial Scala application, consider a web site based on a Scala-based web framework such as **Play Framework**, **Scalatra**, **Xitrum**, or **Lift**. Suppose part of the web application requires plotting forecasted temperature data. Rather than looking for a Scala library to obtain the weather forecasts and another library for plotting, the developer might want to leverage knowledge of the **darksky** (Rudis 2017), **httr** (Wickham 2017), and **ggplot2** (Wickham 2009) packages in R. One could simply execute an R script to run the desired code and read the result from the disk. Another solution is to connect to R using **Rserve** using its Java client. One could go so far as to set up a server using, for example, RApache (Horner 2013), CGIwithR (Firth 2003), or Shiny (Chang, Cheng, Allaire, Xie, and McPherson 2017), all of which require some initial effort and ongoing maintenance beyond the effort required for the Scala-based web framework itself. In contrast, the marginal cost of incorporating **rscala** is low, requiring only the declaration of the dependency on **rscala** in the project's `build.sbt` file (as shown earlier) and a standard installation of R with the **rscala**, **darksky**, **httr**, and **ggplot2** packages. The web application is hosted here:

https://dahl.byu.edu/software/rscala/temperature/

The source code is available here:

https://github.com/dbdahl/rscala-example-temperature/

Figure 1 shows a screenshot of the application.

Apache Spark, a cluster-computing framework for massive datasets, is another example of a Scala application that might benefit from access to R. **Spark** provides an application programming interface to Scala, Java, R, and Python. R users who are not already familiar with Scala would be best served by accessing **Spark** from R using a dedicated package such as **sparklyr** or **sparkr**. Scala developers, however, might prefer to program directly with **Spark**'s Machine Learning Library (MLlib) in Scala and to supplement its functionality with R through **rscala**. Recall that every `RClient` has its own workspace, so several instances can be used to overcome the single-threaded nature of R. One could, for example, use **Apache Commons Pool** to manage a pool of `RClient` objects on each worker node. One potential limitation is the cost of pushing large datasets over the TCP/IP bridge.

4. Case study: Simulation study accelerated with rscala

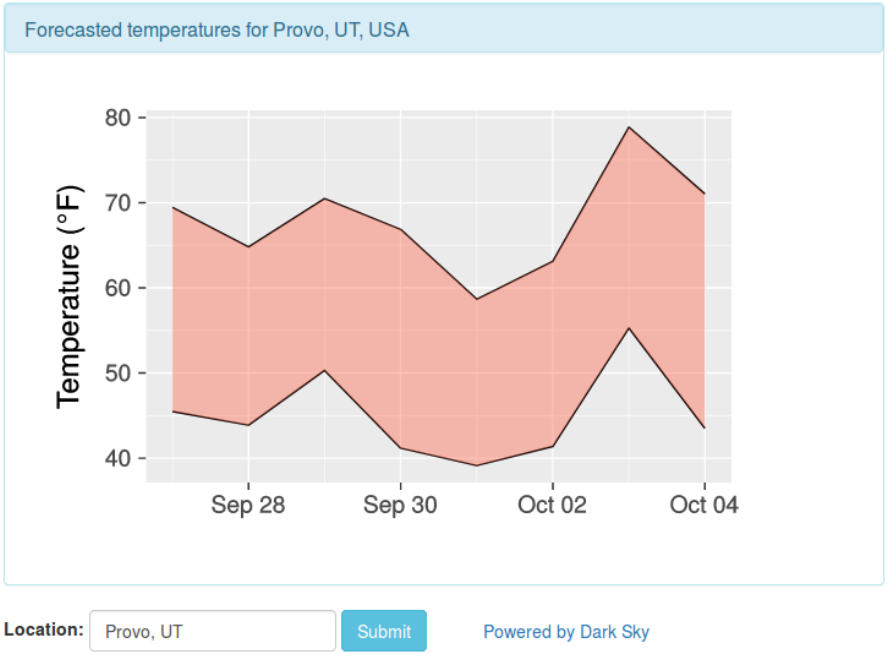

Figure 1: Screenshot of a web application implemented in a Scala-based framework and accessing R packages using **rscala**.

While the previously mentioned **shallot** and **bamboo** packages demonstrate the ability to develop packages based on **rscala**, we now demonstrate the ease with which computationally-intensive statistical procedures can be implemented by embedding Scala code in an R script. The algorithm is embarrassingly parallel and we consider two means of parallelization: one using Scala's `Future` class and the other using R's **parallel** package. By way of comparison, we include a pure R implementation of the same algorithm, and also an implementation that uses inline C++ code via the **Rcpp** package. All four implementations define a function that takes an R sampling function as an argument.

We investigate a simulation study of the coverage probability of a bootstrap confidence interval procedure. Consider a population parameter β_1/β_2, where β_1 and β_2 are population quantiles associated with probabilities p_1 and p_2, respectively. Based on a sample of n observations, a point estimator of the parameter is the ratio of the corresponding sample quantiles and the following bootstrap procedure can be used to find a confidence interval when the population distribution is unspecified. The sample estimate is recorded for each of `nSamples` bootstrap samples. A bootstrap confidence interval is given by (l, u), where l and u are quantiles of the bootstrap sampling distribution associated with $\alpha/2$ and $1 - \alpha/2$, respectively. Although the nominal coverage is $1 - \alpha$, interest lies in computing the actual coverage probability of this bootstrap confidence interval procedure using a Monte Carlo simulation study. `nIntervals` samples from the population are obtained from a user-supplied sampling function. Although the code is general, we sample $n = 100$ observations from the standard normal distribution and set $p_1 = 0.75$ and $p_2 = 0.35$, making $\beta_1/\beta_2 \approx -1.75$. We use `nIntervals` $= 10{,}000$ Monte Carlo replicates, each having `nSamples` $= 10{,}000$ bootstrap samples.

Machine	Implementation	Min.	Q_1	Mean	Median	Q_3	Max.
Ubuntu 8 cores	Pure R	1858.1	1863.9	1875.9	1867.9	1878.9	1943.8
	Rcpp	105.1	105.3	107.3	105.6	106.5	118.6
	rscala #1	81.8	82.0	82.5	82.2	82.5	84.8
	rscala #2	70.2	70.4	71.4	70.7	70.9	77.9
Ubuntu 56 cores	Pure R	444.1	449.8	450.8	451.3	452.4	456.1
	Rcpp	19.7	19.6	19.7	19.8	19.8	20.0
	rscala #1	46.9	47.7	48.0	48.0	48.6	49.0
	rscala #2	14.5	14.6	16.6	14.8	15.0	32.2
Mac 8 cores	Pure R						
	Rcpp	136.9	137.1	139.3	139.8	140.7	141.8
	rscala #1	93.4	94.2	94.7	94.9	95.3	95.8
	rscala #2	91.2	92.5	93.4	92.6	92.7	101.8
Windows 8 cores	Pure R						
	Rcpp	184.8	185.0	187.6	185.3	186.1	201.8
	rscala #1	108.9	109.1	109.3	109.1	109.5	110.1
	rscala #2	106.7	107.3	107.8	107.9	108.0	109.9

Table 3: Elapsed time (in seconds) for the four implementations of the bootstrap simulation study executed on four different machines. Overall, the second **rscala** implementation had the fastest execution times.

The four implementations are available in the Appendix, in the package, and at `https://raw.githubusercontent.com/dbdahl/rscala/master/R/rscala/inst/doc/bootstrap-coverage.R`. The R implementation is the shortest and the **rscala** implementation is somewhat more concise than the **Rcpp** implementation. The pure R iterates using apply functions. The **Rcpp** implementation is written in C style. The pure R, **Rcpp**, and second **rscala** implementations use the **parallel** package to harness all available cores, whereas the first **rscala** implementation uses Scala's `Future` class for parallelism. In the first **rscala** implementation, a single instance of `RClient` is used by multiple JVM threads to call back to the single R instance when sampling the data. On machines with many cores, having each thread wait its turn to access the R instance will likely slow down the execution. In the second **rscala** implementation, each CPU core has a separate R instance with a corresponding `RClient`.

We tested on four machines running Ubuntu 16.04 with 8 cores, Ubuntu 16.04 with 56 cores, Mac Sierra with 8 cores, and Windows 10 with 8 cores. R was compiled from source for the Ubuntu machines and was installed from binaries downloaded from CRAN for the Mac and Windows machines. All machines ran R 3.4.1, Scala 2.12.3, Java 8, **Rcpp** 0.12.12, and a prerelease version of **rscala** 2.3.1.

Elapsed times (in seconds) for 10 replications of the simulation study are found in Table 3. For the sake of expediency, the pure R implementation was only run on the Ubuntu machines. The pure R implementation ran more than 26 times slower than the fastest implementation. The second **rscala** implementation (which uses the **parallel** package) was the fastest overall. The first **rscala** implementation was close behind, except on the Ubuntu machine with 56

cores, which illustrates the difficulty of sharing a single R instance across all of the cores. The **Rcpp** implementation is generally slower than the **rscala** implementations, but still much faster than the pure R implementation.

5. Conclusion

This paper introduced the **rscala** software to bridge R and Scala. The software allows a user to leverage their skills in both languages and to utilize libraries and exploit strengths in each language. For example, R users can implement computationally intensive algorithms in Scala, write R packages based on Scala, and access Scala libraries from R. Scala programmers can take advantage of R's tools for data analysis and graphics from within a Scala application.

We are exploring two possible features to improve the package. The first would allow embedded Scala computations to be interrupted by the R user without destroying the TCP/IP bridge. The second feature would permit R and Scala to run on separate machines.

Acknowledgements

The author's work on this paper was supported by NIH NIGMS R01 GM104972. The author thanks the CRAN maintainers for their excellent service. The author also thanks the following students for valuable feedback on the software and paper: Floid Gilbert, Brandon Carter, Deepthi Uppalapati, Scott Ferguson, and Richard Payne.

References

Bell ET (1938). "The iterated exponential integers." *Annals of Mathematics*, **39**, 539–557. URL http://www.jstor.org/stable/1968633?seq=1#page_scan_tab_contents.

Bugnion P (2016). *Scala for Data Science*. Packt Publishing. URL https://www.amazon.com/Scala-Data-Science-Pascal-Bugnion-ebook/dp/B011V2NPYI?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B011V2NPYI.

Casella G, Moreno E, Girón FJ (2014). "Cluster Analysis, Model Selection, and Prior Distributions on Models." *Bayesian Anal.*, **9**(3), 613–658. doi:10.1214/14-BA869. URL http://dx.doi.org/10.1214/14-BA869.

Chang W, Cheng J, Allaire J, Xie Y, McPherson J (2017). ***shiny**: Web Application Framework for R*. R package version 1.0.4, URL https://CRAN.R-project.org/package=shiny.

Dahl DB (2017a). ***bamboo**: Protein Secondary Structure Prediction Using the Bamboo Method*. R package version 0.9.18, URL https://CRAN.R-project.org/package=bamboo.

Dahl DB (2017b). ***shallot**: Random Partition Distribution Indexed by Pairwise Information*. R package version 0.3.2, URL https://CRAN.R-project.org/package=shallot.

Dahl DB (2017c). ***rscala**: Bi-Directional Interface Between R and Scala with Callbacks*. R package version 2.3.1, URL https://CRAN.R-project.org/package=rscala.

Dahl DB, Day R, Tsai JW (2017). "Random Partition Distribution Indexed by Pairwise Information." *Journal of the American Statistical Association*, **in press**. doi:10.1080/01621459.2016.1165103. URL http://dx.doi.org/10.1080/01621459.2016.1165103.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. URL http://www.jstatsoft.org/v40/i08/.

Firth D (2003). "**CGIwithR**: Facilities for processing web forms using R." *Journal of Statistical Software*, **8**, 1–8. R package version 0.50, URL http://www.omegahat.org/CGIwithR/.

Gouy I (2017). "The Computer Language Benchmarks Game." http://benchmarksgame.alioth.debian.org/u32/scala.html. Accessed: 2017-08-08.

Horner J (2013). ***rApache**: Web application development with R and **Apache***. URL http://www.rapache.net/.

Jancauskas V (2016). *Scientific Computing with Scala*. Packt Publishing. URL https://www.amazon.com/Scientific-Computing-Scala-Vytautas-Jancauskas-ebook/dp/B01ARXUY78?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B01ARXUY78.

Li Q, Dahl DB, Vannucci M, Joo H, Tsai JW (2014). "Bayesian Model of Protein Primary Sequence for Secondary Structure Prediction." *PLOS ONE*, **9**(10), 1–12. doi:10.1371/journal.pone.0109832. URL https://doi.org/10.1371/journal.pone.0109832.

Nicolas PR (2014). *Scala for Machine Learning*. Packt Publishing. URL https://www.amazon.com/Scala-Machine-Learning-Patrick-Nicolas-ebook/dp/B00R6585KO?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B00R6585KO.

Odersky M, Spoon L, Venners B (2016). *Programming in Scala: Updated for Scala 2.12*. 3rd edition. Artima Press. ISBN 0981531687. URL https://www.amazon.com/Programming-Scala-Updated-2-12/dp/0981531687?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0981531687.

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

Pfeffer A (2016). *Practical Probabilistic Programming*. Manning Publications. ISBN 1617292338. URL https://www.amazon.com/Practical-Probabilistic-Programming-Avi-Pfeffer/dp/1617292338?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1617292338.

Premraj R (2015). ***mailR**: A Utility to Send Emails from R*. R package version 0.4.1, URL https://CRAN.R-project.org/package=mailR.

R Core Team (2017). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rudis B (2017). *darksky: Tools to Work with the 'Dark Sky' 'API'*. R package version 1.3.0, URL https://CRAN.R-project.org/package=darksky.

Satman MH (2014). "RCaller: A Software Library for Calling R from Java." *British Journal of Mathematics & Computer Science*, **4**(15), 2188–2196.

Sklyar O, Murdoch D, Smith M, Eddelbuettel D, Francois R, Soetaert K (2015). ***inline**: Functions to Inline C, C++, Fortran Function Calls from R*. R package version 0.3.14, URL https://CRAN.R-project.org/package=inline.

The Apache Software Foundation (2017). ***commonsMath**: JAR Files of the Apache Commons Mathematics Library*. R package version 1.0.0, URL https://CRAN.R-project.org/package=commonsMath.

Urbanek S (2013). ***Rserve**: Binary R server*. R package version 1.7-3, URL https://CRAN.R-project.org/package=Rserve.

Urbanek S (2016). ***rJava**: Low-Level R to Java Interface*. R package version 0.9-8, URL https://CRAN.R-project.org/package=rJava.

Wickham H (2009). ***ggplot2**: Elegant Graphics for Data Analysis*. Springer-Verlag New York. ISBN 978-0-387-98140-6. URL http://ggplot2.org.

Wickham H (2017). *httr: Tools for Working with URLs and HTTP*. R package version 1.3.1, URL https://CRAN.R-project.org/package=httr.

Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ, Ghodsi A, Gonzalez J, Shenker S, Stoica I (2016). "Apache Spark: A Unified Engine for Big Data Processing." *Commun. ACM*, **59**(11), 56–65. ISSN 0001-0782. doi:10.1145/2934664. URL http://doi.acm.org/10.1145/2934664.

Appendix A.

```
#### Code for Section 4. "Case Study: Simulation Study Accelerated with rscala"

makeConfidenceInterval <- function(p, n) {
  me <- qnorm(0.975) * sqrt( p * ( 1 - p ) / n )
  c(estimate = p, lower = p - me, upper = p + me)
}

prob1   <- 0.75
prob2   <- 0.35
truth   <- qnorm(prob1) / qnorm(prob2)
n       <- 100
alpha   <- 0.05


#### rscala implementation #1

library(rscala)
scala()

coverage.rscala1 <- function(sampler=NULL, n=0L, truth=0, prob1=0.0, prob2=0.0,
                             nSamples=1000L, alpha=0.05, nIntervals=1000L) {
  coverage <- s %!% '
    import scala.util.Random
    import scala.concurrent.{Await, Future}
    import scala.concurrent.ExecutionContext.Implicits.global

    def quantile(sorted: Array[Double], p: Double) = {
      val i = ((sorted.length-1)*p).asInstanceOf[Int]
      val delta = (sorted.length-1)*p - i
      ( 1 - delta ) * sorted(i) + delta * sorted(i+1)
    }

    def statistic(x: Array[Double]) = {
      scala.util.Sorting.quickSort(x)
      quantile(x,prob1) / quantile(x,prob2)
    }

    def resample(x: Array[Double], rng: Random) = Array.fill(x.length) {
      x(rng.nextInt(x.length))
    }

    def ciContains(x: Array[Double], rng: Random) = {
      val bs = Array.fill(nSamples) { statistic(resample(x, rng)) }
      scala.util.Sorting.quickSort(bs)
      ( quantile(bs, alpha/2) <= truth ) && ( truth <= quantile(bs, 1-alpha/2) )
    }

    Await.result( Future.sequence( List.fill(nIntervals) {
      val dataset = R.invokeD1(sampler, n)
      val rng = new Random(R.invokeI0("runif", 1, -Int.MaxValue, Int.MaxValue))
      Future { ciContains(dataset, rng) }
    }), concurrent.duration.Duration.Inf).count(identity) / nIntervals.toDouble
  '
  makeConfidenceInterval(coverage, nIntervals)
}
```

```
56
57
58  #### All of the remaining implementation use the parallel package.
59
60  library(parallel)
61  cluster <- makeCluster(detectCores())
62
63
64  #### rscala implementation #2
65
66  clusterEvalQ(cluster, {
67    library(rscala)
68    scala()
69    ciContains.rscala2 <- function(sampler=NULL, n=0L, truth=0, prob1=0.0, prob2=0.0,
70                                   nSamples=1000L, alpha=0.05) {
71      s %!% '
72        def quantile(sorted: Array[Double], p: Double) = {
73          val i = ((sorted.length-1)*p).asInstanceOf[Int]
74          val delta = (sorted.length-1)*p - i
75          ( 1 - delta ) * sorted(i) + delta * sorted(i+1)
76        }
77
78        def statistic(x: Array[Double]) = {
79          scala.util.Sorting.quickSort(x)
80          quantile(x,prob1) / quantile(x,prob2)
81        }
82
83        def resample(x: Array[Double], rng: scala.util.Random) = Array.fill(x.length) {
84          x(rng.nextInt(x.length))
85        }
86
87        val x = R.invokeD1(sampler, n)
88        val rng = new scala.util.Random(R.invokeI0("runif", 1, -Int.MaxValue, Int.MaxValue))
89        val bs = Array.fill(nSamples) { statistic(resample(x, rng)) }
90        scala.util.Sorting.quickSort(bs)
91        ( quantile(bs, alpha/2) <= truth ) && ( truth <= quantile(bs, 1-alpha/2) )
92      '
93    }
94  })
95
96  coverage.rscala2 <- function(sampler, n, truth, prob1, prob2, nSamples, alpha, nIntervals) {
97    clusterExport(cluster, c("sampler","n","truth","prob1","prob2","nSamples","alpha"),
98      envir=environment())
99    coverage <- mean(parSapply(cluster, 1:nIntervals, function(i) {
100     ciContains.rscala2(sampler, n, truth, prob1, prob2, nSamples, alpha)
101   }))
102   makeConfidenceInterval(coverage, nIntervals)
103 }
104
105
106 #### Pure R implementation
107
108 coverage.pureR <- function(sampler, n, truth, prob1, prob2, nSamples, alpha, nIntervals) {
109   statistic <- function(x) {
110     q <- quantile(x, probs = c(prob1, prob2))
111     q[1] / q[2]
112   }
```

```
  ciContains.pureR <- function(x) {
    samples <- sapply(1:nSamples, function(i) {
      statistic(sample(x, replace=TRUE))
    })
    ci <- quantile(samples, probs = c(alpha/2, 1-alpha/2))
    ( ci[1] <= truth ) && ( truth <= ci[2] )
  }
  clusterExport(cluster, c("sampler","n","truth","prob1","prob2","nSamples","alpha"),
    envir = environment())
  coverage <- mean(parSapply(cluster, 1:nIntervals, function(i) {
    ciContains.pureR(sampler(n))
  }))
  makeConfidenceInterval(coverage, nIntervals)
}


#### Rcpp implementation

clusterEvalQ(cluster, { # Don't count compile timing when benchmarking Rcpp.
  library(Rcpp)
  sourceCpp(code="
    #include <Rcpp.h>
    using namespace Rcpp;

    double quantile(double *sorted, int length, double p) {
      int i = (int) ((length-1)*p);
      double delta = (length-1)*p - i;
      return ( 1 - delta ) * sorted[i] + delta * sorted[i+1];
    }

    int compare_double(const void* a, const void* b) {
      double aa = *(double*)a;
      double bb = *(double*)b;
      if ( aa == bb ) return 0;
      return aa < bb ? -1 : 1;
    }

    double statistic(double *x, int length, double prob1, double prob2) {
      qsort(x, length, sizeof(double), compare_double);
      return quantile(x, length, prob1) / quantile(x, length, prob2);
    }

    double *resample(double *x, int length) {
      double *y = (double*) malloc(length*sizeof(double));
      for ( int i=0; i<length; i++ ) y[i] = x[(int)(Rf_runif(0,1)*length)];
      return y;
    }

    // [[Rcpp::export]]
    bool ciContains(NumericVector data, double truth,
                    double prob1, double prob2, int nSamples, double alpha) {
      double *y = (double*) malloc(nSamples*sizeof(double));
      for ( int i=0; i<nSamples; i++ ) {
        int length = data.size();
        double *z = resample(data.begin(), length);
        y[i] = statistic(z, length, prob1, prob2);
        free(z);
```

```
      }
      qsort(y, nSamples, sizeof(double), compare_double);
      bool result =  ( quantile(y, nSamples,   alpha/2) <= truth ) &&
                     ( quantile(y, nSamples, 1-alpha/2) >= truth );
      free(y);
      return result;
    }
  ")
})

coverage.Rcpp <- function(sampler, n, truth, prob1, prob2, nSamples, alpha, nIntervals) {
  clusterExport(cluster, c("sampler","n","truth","prob1","prob2","nSamples","alpha"),
    envir=environment())
  coverage <- mean(parSapply(cluster, 1:nIntervals, function(i) {
    ciContains(sampler(n), truth, prob1, prob2, nSamples, alpha)
  }))
  makeConfidenceInterval(coverage, nIntervals)
}


#### Benchmarks

library(microbenchmark)
engine <- function(nSamples, nIntervals) microbenchmark(
  pureR.   = coverage.pureR(  rnorm, n, truth, prob1, prob2, nSamples, alpha, nIntervals),
  Rcpp.    = coverage.Rcpp(   rnorm, n, truth, prob1, prob2, nSamples, alpha, nIntervals),
  rscala1. = coverage.rscala1(rnorm, n, truth, prob1, prob2, nSamples, alpha, nIntervals),
  rscala2. = coverage.rscala2(rnorm, n, truth, prob1, prob2, nSamples, alpha, nIntervals),
  times=10)

engine(nSamples = 10000L, nIntervals = 10000L)
```

Affiliation:

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

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* 2017-08-22
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd