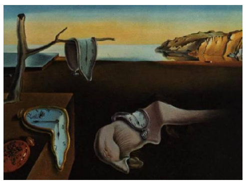


Journal of Statistical Software

January 2009, Volume 29, Issue 4. *http://www.jstatsoft.org/*

RinRuby: Accessing the R Interpreter from Pure Ruby

David B. Dahl
Texas A&M University

Scott Crawford
Texas A&M University

Abstract

RinRuby is a Ruby library that integrates the R interpreter in Ruby, making R's statistical routines and graphics available within Ruby. The library consists of a single Ruby script that is simple to install and does not require any special compilation or installation of R. Since the library is 100% pure Ruby, it works on a variety of operating systems, Ruby implementations, and versions of R. **RinRuby**'s methods are simple, making for readable code. This paper describes **RinRuby** usage, provides comprehensive documentation, gives several examples, and discusses **RinRuby**'s implementation. The latest version of **RinRuby** can be found at the project website: `http://rinruby.ddahl.org/`.

Keywords: R, Ruby, JRuby, Java, **RinRuby**.

1. Introduction

Scripting languages such as Ruby, Python, Perl, and PHP are increasingly popular since they can greatly decrease development time compared to traditional languages, such as C, C++, and Java. Although many variations exist, scripting languages are high-level programming languages that typically avoid explicit type declarations, interpret or compile code at runtime, and focus on simplicity and productivity rather than raw execution speed. Scripting languages have been particularly successful in tasks such as data extraction, web development, prototyping, report generation, and combining existing software to accomplish a task.

Ruby is a dynamic scripting language "with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write" (Flanagan and Matsumoto 2008). Ruby supports features such as pure object orientation, closures, and mix-ins. Several implementations are available, the most mature being the reference C implementation, JRuby (which runs on the Java Virtual Machine), IronRuby (which runs on the .NET Framework), and Rubinius (an alternative implementation written in Ruby and C). Unfortunately statistical analysis routines and graphing abilities are quite limited in Ruby.

R is a scripting language and environment developed by statisticians for statistical computing and graphics with a large library of routines (R Development Core Team 2008). R has many contributors and a large user base which increases confidence in the correctness of the implementation. The graphing abilities of R are excellent.

This paper describes the **RinRuby** software, a 100% pure Ruby library that provides a simple but effective bridge to R from Ruby. Being 100% pure Ruby, **RinRuby** does not need to be recompiled with each incremental release of R and Ruby. It allows a statistician to leverage R's familiar and comprehensive statistical computing and graphics abilities in the powerful Ruby scripting language. (Note that **RinRuby** does not provide access *to* Ruby *from* R.)

RinRuby's design allows R to be accessed from Ruby on any implementation of Ruby using a standard installation of R on any operating system capable of running R and Ruby (including Linux, Mac OS X, and Microsoft Windows). This means there is no need to install Ruby or R with special options. Using **RinRuby** in JRuby, for example, allows for seamless integration of Ruby, Java, and R code in one application.

The paper is organized as follows. Section 2 introduces **RinRuby** basics, including installation and typical usage. An example using the Gettysburg Address is given in Section 3. Section 4 discusses approaches to making R accessible in a scripting language and details the technique used by **RinRuby**. Comprehensive documentation is provided in Section 5, while Section 6 discusses a few caveats related to **RinRuby** usage. The appendix contains two examples: our **RinRuby** translation of Tim Churches' demonstration of **RPy** (a similar program that makes R accessible within Python, see Moreira and Warnes 2008) and an example involving simple linear regression. The scripts for all the examples are available online along with the paper.

2. Using RinRuby

2.1. Installation

A prerequisite for **RinRuby** is a working installation of R, but no special compilation flags, installation procedures, or packages are needed for R. If using the RubyGems system, **RinRuby** can be installed by simply executing the following at the operating system's shell prompt (denoted `$`):

```
$ gem install rinruby
```

This will download and install the latest version of **RinRuby** from RubyForge (`http://rubyforge.org/`), an archive of Ruby extensions analogous to the Comprehensive R Archive Network for R. The equivalent call for JRuby is:

```
$ jruby -S gem install rinruby
```

If RubyGems is not available, the latest version of the `rinruby.rb` script can be downloaded from the **RinRuby** webpage (`http://rinruby.ddahl.org/`) and placed in a directory in Ruby's search path (as given by the array `$:`).

2.2. Executing R commands

Regardless of the installation method, **RinRuby** is invoked within a Ruby script (or the interactive "irb" prompt, denoted >>) using:

```
>> require "rinruby"
```

The previous statement reads the definition of the **RinRuby** class into the current Ruby interpreter and creates an instance of the **RinRuby** class named R. The eval instance method passes R commands (contained in the supplied string) and prints the output or displays any resulting plots. For example:

```
>> sample_size = 10
>> R.eval "x <- rnorm(#{sample_size})"
>> R.eval "summary(x)"
>> R.eval "sd(x)"
```

produces the following:

```
   Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-1.88900 -0.84930 -0.45220 -0.49290 -0.06069  0.78160

[1] 0.7327981
```

This example uses a string substitution to make the argument of the first eval method equivalent to x <- rnorm(10). The example uses three invocations of the eval method, but a single invoke is possible using a here document:

```
>> sample_size = 10
>> R.eval <<EOF
      x <- rnorm(#{sample_size})
      summary(x)
      sd(x)
   EOF
```

2.3. Pulling data from R to Ruby

Data is copied from R to Ruby using the pull method or a short-hand equivalent. The R object x defined previously can be copied to the Ruby object copy_of_x as follows:

```
>> copy_of_x = R.pull "x"
>> puts copy_of_x
```

which produces the following: (note only the first and last lines are shown)

```
-0.376404489256671
⋮
0.781602719849499
```

RinRuby also supports a convenient short-hand notation when the argument to `pull` is simply a previously-defined R variable (whose name conforms to Ruby's requirements for method names). For example:

```
>> copy_of_x = R.x
```

The explicit `pull` method, however, can take an arbitrary R statement. For example:

```
>> summary_of_x = R.pull "as.numeric(summary(x))"
>> puts summary_of_x
```

produces the following :

```
-1.889
-0.8493
-0.4522
-0.4929
-0.06069
0.7816
```

Notice the use above of the `as.numeric` function in R. This is necessary since the `pull` method only supports R vectors which are `numeric` (i.e., integers or doubles) or `character` (i.e., strings). Data in other formats must be coerced when copying to Ruby.

2.4. Assigning data from Ruby to R

Data is copied from Ruby to R using the `assign` method or a short-hand equivalent. For example:

```
>> names = ["Lisa", "Teasha", "Aaron", "Thomas"]
>> R.assign "people", names
>> R.eval "sort(people)"
```

produces the following :

```
[1] "Aaron"  "Lisa"   "Teasha" "Thomas"
```

The short-hand equivalent to the `assign` method is simply:

```
>> R.people = names
```

As with the short-hand notation for `pull`, some care is needed when using the short-hand of the `assign` method since the label (i.e., `people` in this case) must be a valid method name in Ruby. For example, `R.copy.of.names = names` will not work, but `R.copy_of_names = names` is permissible.

The `assign` method supports Ruby variables of type `Fixnum` (i.e., integer), `Bignum` (i.e. integer), `Float` (i.e., double), `String`, and arrays of one of those four fundamental types. Data in other formats must be coerced when copying to R. Note that `Fixnum` or `Bignum` values that exceed the capacity of R's integers are silently converted to doubles. Data in other formats must be coerced when copying to R.

When assigning an array containing differing types of variables, **RinRuby** will follow R's conversion conventions. An array that contains any `String`s will result in a character vector in R. If the array does not contain any `String`s, but it does contain a `Float` or a large `Integer` (in absolute value), then the result will be a numeric vector of `Double`s in R. If there are only `Integer`s that are sufficiently small (in absolute value), then the result will be a numeric vector of `Integer`s in R.

3. Demonstration using the Gettysburg Address

The `eval`, `assign`, and `pull` methods are demonstrated in an example using Lincoln's Gettysburg Address. The code is in Table 1 and the output is in Table 2. Two more extensive examples are given in the Appendix.

Ruby code counts the number of occurrences of each word in the Gettysburg Address and filters out the words occurring less than three times or shorter than four letters. R code—through the **RinRuby** library—produces a bar plot of the most frequent words and computes the correlation between the length of a word and the usage frequency. Finally, the correlation is printed by Ruby.

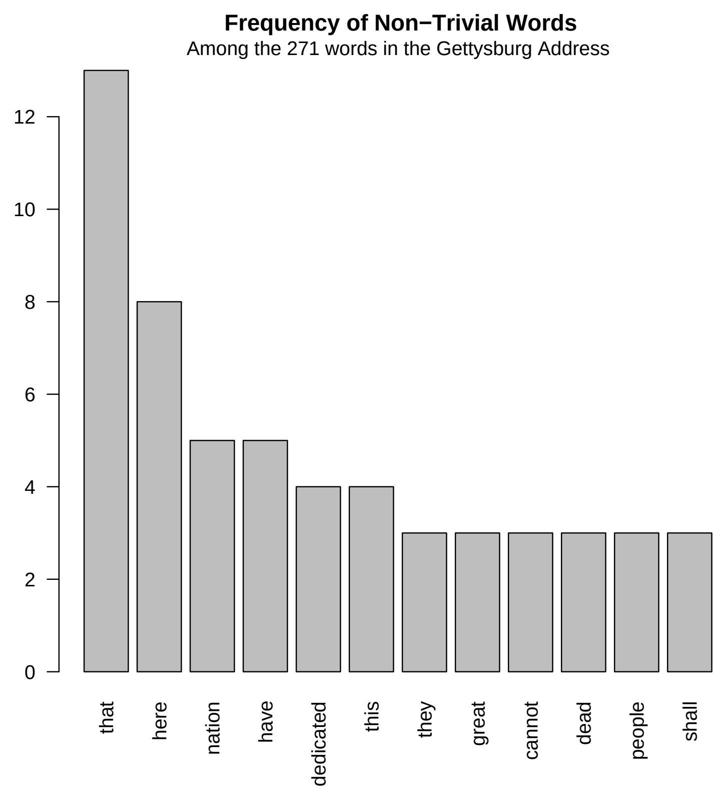
```
tally = Hash.new(0)
File.open('gettysburg.txt').each_line do |line|
  line.downcase.split(/\W+/).each { |w| tally[w] += 1 }
end
total = tally.values.inject { |sum,count| sum + count }
tally.delete_if { |key,count| count < 3 || key.length < 4 }

require "rinruby"
R.keys, R.counts = tally.keys, tally.values

R.eval <<EOF
  names(counts) <- keys
  barplot(rev(sort(counts)), main = "Frequency of Non-Trivial Words", las = 2)
  mtext("Among the #{total} words in the Gettysburg Address", 3, 0.45)
  rho <- round(cor(nchar(keys), counts), 4)
EOF

puts "The correlation between length and frequency of words is #{R.rho}."
```

Table 1: Demonstration of **RinRuby** using Lincoln's Gettysburg Address.



The correlation between length and frequency of words is -0.2779.

Table 2: Output from the Gettysburg example.

4. Techniques to access R

4.1. Techniques used by existing software

In this section, we review several approaches which could embed R in a scripting language and we discuss the approach used by **RinRuby**. Perhaps the most obvious way to use R in a Ruby script is to: (1) Generate an R script using Ruby code, (2) Run the R script in batch mode using Ruby's `system` method, and (3) Retrieve computations by parsing the resulting `Rout` file in Ruby. This method is simple in principle but has at least two deficiencies. First, passing results and data can require a fair amount of problem-specific coding. The **RRb** (Nakao 2003) package attempts to ease this difficulty, but currently it is only available for Linux. The second problem is that each new computation in R requires starting a new instance of the R interpreter.

Another approach is to write a native-language (i.e., C) extension for Ruby that dynamically links to R's shared library. This is the approach taken by **RSRuby** (Gutteridge 2008) for Ruby, **RSPerl** (Temple Lang 2007) for Perl, and **RPy** (Moreira and Warnes 2008) for Python. **RSRuby** is making rapid progress towards seamless integration of Ruby and R and provides very fast data access. The disadvantage comes in the fact that the extension is closely tied to a particular operating system, Ruby implementation, and version of R. For example, users must compile **RSRuby** themselves or wait for maintainers to provide binaries for recently released versions of R and Ruby. Further, **RSRuby** is currently not available for Windows. Lastly, the extension is tied to the particular implementation of the Ruby language and is only a proof-of-concept for alternative implementations of the Ruby language (e.g., JRuby, IronRuby, and Rubinius).

Another approach to using R within Ruby is the client/server model over TCP/IP sockets. **RServe** (Urbanek 2008) uses this method, where R runs as the server for a client application. This approach is quite efficient and very robust across different systems, platforms, and versions. Unfortunately only Java and C++ clients are currently implemented; there is no Ruby implementation.

4.2. Technique used by RinRuby

RinRuby makes use of pipes and TCP/IP sockets to implement its functionality. When the **RinRuby** module is loaded via the `require "rinruby"` statement, RinRuby opens an operating system pipe to the R interpreter running on the same machine using Ruby's `IO.popen` method. The pipe stays open until the `quit` method is called or Ruby exits. When **RinRuby**'s `eval` method is called, the statement is passed to R through this pipe. Additionally, another statement is passed to R that prints a flag signaling that R is done executing. **RinRuby** prints all the results from the pipe using Ruby's `puts` method until it sees this signal, at which point the `eval` method returns.

RinRuby passes data between Ruby and R using TCP/IP sockets on the localhost. Ruby acts as the server and R acts as the client. Like the pipe, the socket connection stays open until calling `quit` or exiting Ruby. When pulling data from R to Ruby, **RinRuby** writes R statements to the pipe that causes R to send the data over the socket. Likewise, assigning data from Ruby to R involves sending R statements through the pipe to receive the data. Passing data via sockets instead of the pipe enables **RinRuby** to handle large amounts of data quickly. It also avoids rounding issues inherent when converting decimals to text and back to decimals again. Since all the R code needed to implement **RinRuby**'s functionality is sent via the pipe, there is no need to install any **RinRuby**-specific packages in R.

5. Comprehensive documentation

5.1. Instantiating a RinRuby object

RinRuby is invoked within a Ruby script (or the interactive "irb" prompt denoted `>>`) using:

```
>> require "rinruby"
```

This reads the definition of the **RinRuby** class into the current Ruby interpreter and creates an instance of the **RinRuby** class named `R`. An instance of the **RinRuby** class can also be created using the `new` constructor. For example:

```
>> require "rinruby"
>> myr = RinRuby.new
>> myr.eval "rnorm(1)"
```

Any number of independent instances of R can be created in this way.

The following parameters can be passed to the constructor:

```
RinRuby.new(echo = true, interactive = true, executable = nil,
            port_number = 38442, port_width = 1000)
```

- `echo`: By setting the `echo` to `false`, output from R is suppressed, although warnings may still print. This option can be changed later by using the `echo` method which is explained below. The default is `true`.
- `interactive`: When `interactive` is `false`, R is run in non-interactive mode, resulting in plots without an explicit device being written to `Rplots.pdf`. Otherwise, when `interactive` is `true`, such plots are shown on the screen. The default is `true`.
- `executable`: The path of the R executable (which is `R` in Linux and Mac OS X, or `Rterm.exe` in Windows) can be set with the `executable` argument. The default is `nil` which makes **RinRuby** use the registry keys to find the path (on Windows) or use the path defined by `$PATH` (on Linux and Mac OS X).
- `port_number`: This is the smallest port number on the local host that could be used to pass data between Ruby and R. The actual port number used depends on `port_width` described below.
- `port_width`: **RinRuby** will randomly select a uniform number between `port_number` and `port_number` + `port_width` − 1 (inclusive) to pass data between Ruby and R. If the randomly selected port is not available, **RinRuby** will continue selecting random ports until it finds one that is available. By setting `port_width` to 1, **RinRuby** will wait until `port_number` is available. The default `port_width` is `1000`.

It may be desirable to change the parameters to the instance of R, but still name the object `R`. In that case the old instance of R which was created with the `require "rinruby"` statement should be closed first using the `quit` method which is explained below. Unless the previous instance is killed, it will continue to use system resources until exiting Ruby. The following shows an example that explicitly sets the echo argument:

```
>> require "rinruby"
>> R.quit
>> R = RinRuby.new(false)
```

5.2. The "`echo`" method

The `echo` method controls whether the `eval` method displays output and warnings from R. The `echo` method has two parameters:

```
echo(enable = nil, stderr = nil)
```

- `enable`: Setting enable to `false` will turn all output off until the echo method is used again with enable equal to `true`. The default is `nil`, which will return the current setting.
- `stderr`: Setting stderr to `true` will force messages, warnings, and errors from R to be routed through `stdout`. Using `stderr` redirection is typically not needed for the C implementation of Ruby and is thus not enabled by default for this implementation. It is typically necessary for JRuby and is enabled by default in this case. This redirection works well in practice but it can lead to interleaving output which may confuse **RinRuby**. In such cases, `stderr` redirection should not be used. Echoing must be enabled when using `stderr` redirection.

5.3. The "eval" method

The `eval` method is used to send commands to the R instance. The method has two parameters:

```
eval(string, echo_override = nil)
```

- `string`: The `string` parameter is the code which is to be passed to R, for example, `"hist(gamma(1000,5,3))"`. The `string` can contain many lines by use of a here document. For example:

  ```
  R.eval <<EOF
    x <- rgamma(1000, 5, 3)
    hist(x)
  EOF
  ```

- `echo_override`: This argument allows one to set the echo behavior for this call only. The default for `echo_override` is `nil`, which does not override the current echo behavior.

The return value of the `eval` method is `true` unless the statement could not be parsed as a valid R expression. In this case, an exception is raised. See Section 6 for examples of code that would raise an exception from the `eval` method.

5.4. The "assign" method

The `assign` method is used to send data from Ruby to R and has three parameters:

```
assign(name, value, as_integer = false)
```

- `name`: The name of the R variable.
- `value`: The value the R variable should have. The `assign` method supports Ruby variables of type `Fixnum` (i.e., integer), `Bignum` (i.e., integer), `Float` (i.e., double), `String`, and arrays of one of those three fundamental types. Note that `Fixnum` or `Bignum` values that exceed the capacity of R's integers are silently converted to doubles. Data in other formats must be coerced when copying to R.

The assign method is an alternative to the simplified method, with some additional flexibility. When using the simplified method, the parameters of name and value are automatically used. For example, the code:

```
>> R.test = 144
```

is the same as:

```
>> R.assign("test", 144)
```

The shorthand notation cannot be used to assign a variable named `eval`, `echo`, or another already used method name. **RinRuby** would assume the method was being called, rather than assigning a variable.

When assigning an array containing differing types of variables, **RinRuby** will follow R's conversion conventions. An array that contains any `String`s will result in a character vector in R. If the array does not contain any `String`s, but it does contain a `Float` or a large integer (in absolute value), then the result will be a numeric vector of `Double`s in R. If there are only integers that are sufficiently small (in absolute value), then the result will be a numeric vector of integers in R.

5.5. The "pull" method

The `pull` method is used to pass data from R into Ruby. There are two parameters for the `pull` method:

```
pull(string, singletons = false)
```

- `string`: The name of the variable that should be pulled from R. The `pull` method only supports R vectors which are `numeric` (i.e., integers or doubles) or `character` (i.e., strings). The R value of `NA` is pulled as `nil` into Ruby. Data in other formats must be coerced when copying to Ruby.
- `singletons`: R represents a single number as a vector of length one, but in Ruby it is often more convenient to use a number rather than an array of length one. Setting `singletons = false` will cause the `pull` method to shed the array, while `singletons = true` will return the number or string within an array. The default is `false`.

The `pull` method is an alternative to the simplified form where the parameters are automatically used. For example, the code:

```
>> puts R.test
```

is the same as:

```
>> puts R.pull("test")
```

As is the case with the `assign` method, the shorthand notation cannot be used on a variable which shares a name with another method.

5.6. The "prompt" method

When sending code to Ruby using an interactive prompt (i.e., "irb"), this method will change the Ruby prompt to an R prompt. From the R prompt, commands can be sent to R as if R were run natively. When the user is ready to return to Ruby, then exit() will return the prompt to Ruby. The prompt method is useful when exploring with several lines of code since results are displayed immediately. It should be noted that the prompt command does not work in a script, just Ruby's interactive irb.

The prompt command has two parameters:

```
prompt(regular_prompt = "> ", continue_prompt = "+ ")
```

- regular_prompt: This defines the string used to denote the R prompt.
- continue_prompt: This is the string used to denote R's prompt for an incomplete statement (such as a multiple line loop).

5.7. The "quit" method

The quit method will properly close the bridge between Ruby and R, freeing up system resources. This method does not need to be run when a Ruby script ends. There are no parameters in the quit method.

6. Caveats

The assign method supports Ruby variables and arrays of type Fixnum, Bignum, Float, or String, while the pull method supports R vectors which are numeric or character. There is no technical limitation prohibiting pulling or assigning other data types, and we encourage contributions to extend the data types supported by **RinRuby**.

We recommend using **RinRuby** on R version 2.7.0 or higher, which provides the "--interactive" command line argument for Linux and Mac OS X. If a user has version 2.6.2 or earlier, then the plots will not show up on the screen as usual. A warning will appear saying, "unknown option '--interactive'" and plots will be saved to the Rplots.ps file.

R commands sent by **RinRuby**'s eval method should be complete expressions. If R cannot parse the command as a complete expression, the eval method will raise an exception. For example:

```
>> R.eval "paste('answer, x)"
```

will raise an exception indicating a parse error. Another manifestation of the same problem is in the following code:

```
>> R.x = 2
>> R.eval "for (i in 1:10) {"
>> R.eval "  x <- x + 1"
>> R.eval "}"
```

The code will raise an exception for each of three statements in the `for` loop since none of them can be considered a complete expression alone. There are two ways to avoid this issue. One way is to use the `prompt` method while running Ruby interactively. The other way is to use a here document, as shown:

```
>> R.x = 2
>> R.eval <<EOF
     for(i in 1:10){
       x <- x + 1
     }
   EOF
```

Acknowledgments

The authors gratefully acknowledge Marina Vannucci of Rice University for her generous support of the second author. The authors also thank and the editor Jan de Leeuw as well as the two anonymous referees for their suggestions which substantially improved the paper and RinRuby.

References

Flanagan D, Matsumoto Y (2008). *The Ruby Programming Language*. O'Reilly Media, Inc. ISBN 10: 0-596-51617-7, URL http://www.ruby-lang.org/.

Gutteridge A (2008). "**RSRuby**: A Bridge Between Ruby and the R Interpreted Language." Ruby package version 0.5.1, URL http://rubyforge.org/projects/rsruby/.

Moreira W, Warnes GR (2008). "**RPy**: A Simple and Efficient Access to R from Python." Version 1.0.3, URL http://rpy.sourceforge.net/.

Nakao MC (2003). "**RRb**: A Very Simple Ruby Interface to the R Statistical Computing Language." URL http://sourceforge.net/projects/rrb/.

R Development Core Team (2008). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

Temple Lang D (2007). "**RSPerl**: The R/S-PLUS–Perl Interface." R package version 0.92-1, URL http://www.omegahat.org/RSPerl/.

Urbanek S (2008). ***Rserve**: Binary R Server.* R package version 0.5-2, URL http://CRAN.R-project.org/package=Rserve.

A. RinRuby implementation of an example from Tim Churches

Here we provide a side-by-side comparison of Tim Churches' demonstration of **RPy** for Python (`http://rpy.sourceforge.net/rpy_demo.html`) and our implementation using **RinRuby** for Ruby. The code in Tables 3 and 4 analyzes data on the Old Faithful geyser in Yellowstone National Park. The code illustrates the ease and power of **RinRuby** as well as its methods. The output from both scripts is the same, except that **RinRuby** maintains the order of the summary statistics while **RPy** changes the order. The output shown in Table 5 is from **RinRuby** as is Figure 1.

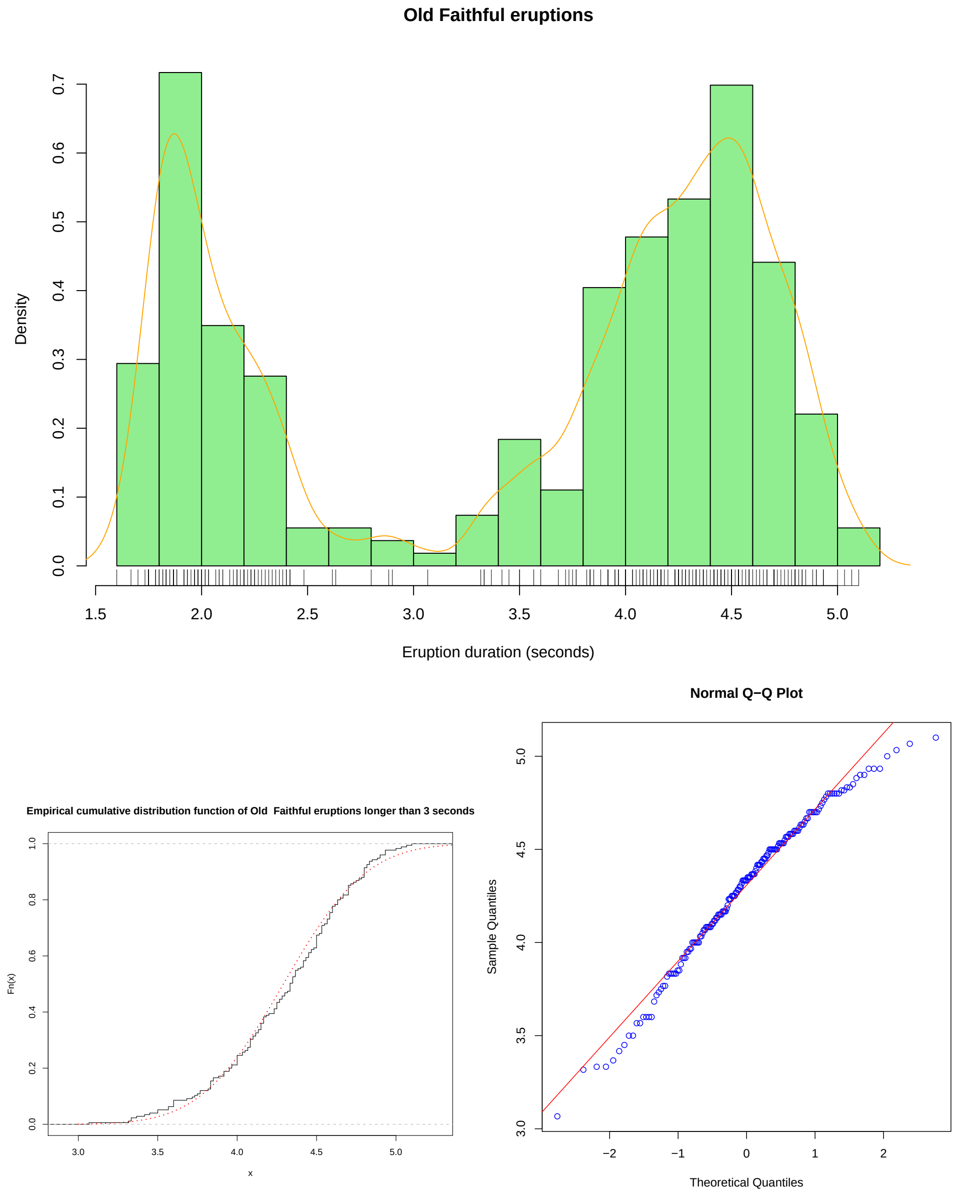


Figure 1: Plots from the Tim Churches example.

```ruby
require "rinruby"

faithful_data = {"eruption_duration"=>[], "waiting_time"=>[]}


for row in File.readlines('faithful.dat')[1..-1]
  splitrow = row.chomp.split
  faithful_data["eruption_duration"] << splitrow[0].to_f
  faithful_data["waiting_time"] << splitrow[1].to_i
end




R.ed = faithful_data["eruption_duration"]
R.eval "edsummary <- summary(ed)"
edsummary = R.pull("as.vector(edsummary)")
keys = R.pull("names(edsummary)")
puts "Summary of Old Faithful eruption duration data"
keys.each_index do |i|
  puts "#{keys[i]}:#{sprintf('%.3f', edsummary[i])}"
end
puts
puts "Stem-and-leaf plot of Old Faithful eruption duration data"
R.eval "stem(ed)"

R.eval <<EOF
  png("faithful_histogram.png", width=10,height=7.5)
  hist(ed, seq(1.6, 5.2, 0.2), prob = 1,col = "lightgreen",
    main = "Old Faithful eruptions", xlab = "Eruption duration (seconds)")
  lines(density(ed, bw = 0.1), col = "orange")
  rug(ed)
  dev.off()
EOF
```

```python
from rpy import *

faithful_data = {"eruption_duration":[],
                 "waiting_time":[]}

f = open('faithful.dat','r')

for row in f.readlines()[1:]: # skip the column header line
  splitrow = row[:-1].split(" ")
  faithful_data["eruption_duration"].append(float(splitrow[0]))
  faithful_data["waiting_time"].append(int(splitrow[1]))

f.close()

ed = faithful_data["eruption_duration"]
edsummary = r.summary(ed)
print "Summary of Old Faithful eruption duration data"
for k in edsummary.keys():
  print k + ": %.3f" % edsummary[k]
print
print "Stem-and-leaf plot of Old Faithful eruption duration data"
print r.stem(ed)


r.png('faithful_histogram.png', width=733, height=550)
r.hist(ed, r.seq(1.6, 5.2, 0.2), prob = 1,col = "lightgreen",
  main = "Old Faithful eruptions", xlab = "Eruption duration (seconds)")
r.lines(r.density(ed, bw = 0.1), col = "orange")
r.rug(ed)
r.dev_off()
```

Table 3: Ruby (left) and Python (right) code for the Tim Churches' example.

```ruby
cutoff = 3
R.long_ed = R.ed.delete_if{ |x| x <= cutoff }
R.eval <<EOF
  png('faithful_ecdf.png', width = 10, height = 7.5)
  # library(stepfun)          # package has been merged into 'stats'
  plot(ecdf(long_ed), do.points=0, verticals = 1,
    main = paste('Empirical cumulative distribution function of Old',
                 ' Faithful eruptions longer than #{cutoff} seconds'))
  x <- seq(3, 5.4, 0.01)
  lines(seq(3, 5.4, 0.01), pnorm(seq(3, 5.4, 0.01),
    mean = mean(long_ed), sd = sqrt(var(long_ed))), lty = 3,
    lwd = 2, col = 'red')
  dev.off()

  png('faithful_qq.png', width = 10, height = 7.5)
  par(pty = "s")
  qqnorm(long_ed,col = "blue")
  qqline(long_ed,col = "red")
  dev.off()
EOF

# R.eval "library('ctest')"   # package has been merged into 'stats'
puts
puts "Shapiro-Wilks normality test of Old Faithful eruptions" +
     " longer than #{cutoff} seconds"
R.eval "sw <- shapiro.test(long_ed)"
puts "W = #{sprintf("%.4f", R.pull("sw$statistic"))}"
puts "p-value = #{sprintf("%.5f", R.pull("sw$p.value"))}"

puts
puts "One-sample Kolmogorov-Smirnov test of Old Faithful eruptions" +
     " longer than #{cutoff} seconds"
R.eval "ks <- ks.test(long_ed, 'pnorm', mean=mean(long_ed)," +
                     "sd = sqrt(var(long_ed)))"
puts "D = #{sprintf("%.4f", R.pull("ks$statistic"))}"
puts "p-value = #{sprintf("%.4f", R.pull("ks$p.value"))}"
puts "Alternative hypothesis: #{R.pull("ks$alternative")}"
puts
```

```python
long_ed = filter(lambda x: x > 3, ed)
r.png('faithful_ecdf.png', width = 733, height = 550)
r.library('stepfun')
r.plot(r.ecdf(long_ed), do_points = 0, verticals = 1, col = "blue",
       main = paste("Empirical cumulative distribution function",
                    " of Old Faithful eruptions longer than 3 seconds")
x = r.seq(3, 5.4, 0.01)
r.lines(r.seq(3, 5.4, 0.01), r.pnorm(r.seq(3, 5.4, 0.01),
  mean = r.mean(long_ed), sd = r.sqrt(r.var(long_ed))), lty = 3,
  lwd = 2, col = "red")
r.dev_off()


r.png('faithful_qq.png', width = 733, height = 550)
r.par(pty = "s")
r.qqnorm(long_ed,col = "blue")
r.qqline(long_ed,col = "red")
r.dev_off()


r.library('ctest')
print
print("Shapiro-Wilks normality test of Old Faithful eruptions" +\
      " longer than 3 seconds")
sw = r.shapiro_test(long_ed)
print "W = %.4f" % sw['statistic']['W']
print "p-value = %.5f" % sw['p.value']

print
print("One-sample Kolmogorov-Smirnov test of Old Faithful eruptions" +\
      " longer than 3 seconds"
ks = r.ks_test(long_ed, "pnorm", mean = r.mean(long_ed),
               sd = r.sqrt(r.var(long_ed)))
print "D = %.4f" % ks['statistic']['D']
print "p-value = %.4f" % ks['p.value']
print "Alternative hypothesis: %s" % ks['alternative']
print
```

Table 4: Ruby (left) and Python (right) code for the Tim Churches' example.

```
Summary of Old Faithful eruption duration data
Min.: 1.600
1st Qu.: 2.163
Median: 4.000
Mean: 3.488
3rd Qu.: 4.454
Max.: 5.100

Stem-and-leaf plot of Old Faithful eruption duration data

  The decimal point is 1 digit(s) to the left of the |

  16 | 070355555588
  18 | 000022233333335577777777888822335777888
  20 | 00002223378800035778
  22 | 0002335578023578
  24 | 00228
  26 | 23
  28 | 080
  30 | 7
  32 | 2337
  34 | 250077
  36 | 0000823577
  38 | 2333335582225577
  40 | 0000003357788888002233555577778
  42 | 03335555778800233333555577778
  44 | 02222335557780000000023333357778888
  46 | 0000233357700000023578
  48 | 00000022335800333
  50 | 0370

Shapiro-Wilks normality test of Old Faithful eruptions longer than 3 seconds
W = 0.9793
p-value = 0.01052

One-sample Kolmogorov-Smirnov test of Old Faithful eruptions longer than
3 seconds
D = 0.0661
p-value = 0.4284
Alternative hypothesis: two-sided
```

Table 5: Output from the Tim Churches' example.

B. A simple linear regression example

As another example of **RinRuby** usage, Table 6 shows the usage of **RinRuby** for simple linear regression. The simulation parameters are defined in Ruby, computations are performed in R, and Ruby reports the results. In a more elaborate application, the simulation parameter could be input from a graphical user interface, the statistical analysis might be more involved, and the results could be an HTML page or PDF report.

```
require "rinruby"

n = 10
beta_0 = 1
beta_1 = 0.25
alpha = 0.05
seed = 23423

R.x = (1..n).entries
R.eval <<EOF
  set.seed(#{seed})
  y <- #{beta_0} + #{beta_1}*x + rnorm(#{n})
  fit <- lm( y ~ x )
  est <- round(coef(fit),3)
  pvalue <- summary(fit)$coefficients[2, 4]
EOF

puts "E(y|x) ~= #{R.est[0]} + #{R.est[1]} * x"
if R.pvalue < alpha
  puts "Reject the null hypothesis and conclude that x and y are related."
else
  puts "There is insufficient evidence to conclude that x and y are related."
end
\end{verbatim}
%\vspace{4ex}
```

```
E(y|x) ~= 1.264 + 0.273 * x
Reject the null hypothesis and conclude that x and y are related.
```

Table 6: Demonstration of **RinRuby** for regression.

Affiliation:

David B. Dahl
Assistant Professor
Department of Statistics
Texas A&M University
3134 TAMU
College Station, Texas 77840, United States of America
E-mail: dahl@stat.tamu.edu
URL: http://www.stat.tamu.edu/~dahl/

Scott Crawford
Ph.D. Candidate
Department of Statistics
Texas A&M University
3134 TAMU
College Station, Texas 77840, United States of America
E-mail: crawford@stat.tamu.edu
URL: http://www.stat.tamu.edu/~crawford/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 29, Issue 4 *Submitted:* 2008-07-03
January 2009 *Accepted:* 2008-11-26