A First Session in R & RStudio

Matthew D. Turner
Department of Psychology, Georgia State University
mturner46@gsu.edu

Introduction

This document covers the examples from the first demonstration of RStudio in the workshop, you can use it to help replicate the examples on your own or to look up any terms you forget.

This document will mention the highlights, but it is not a complete record of the demonstration! Please see the original code for all of the examples. This supplement is designed to assist first time users who may need more details or who want to learn the material on their own.

All of the commands used in the demonstration are provided in the R file:

`W01_FirstSessionInRandRStudio.R`

This file is part of the materials provided for download from the USB sticks circulating at the workshop. Make sure to open that file in RStudio before you read this so you can work along as you read.

All the materials for the workshop are available at:

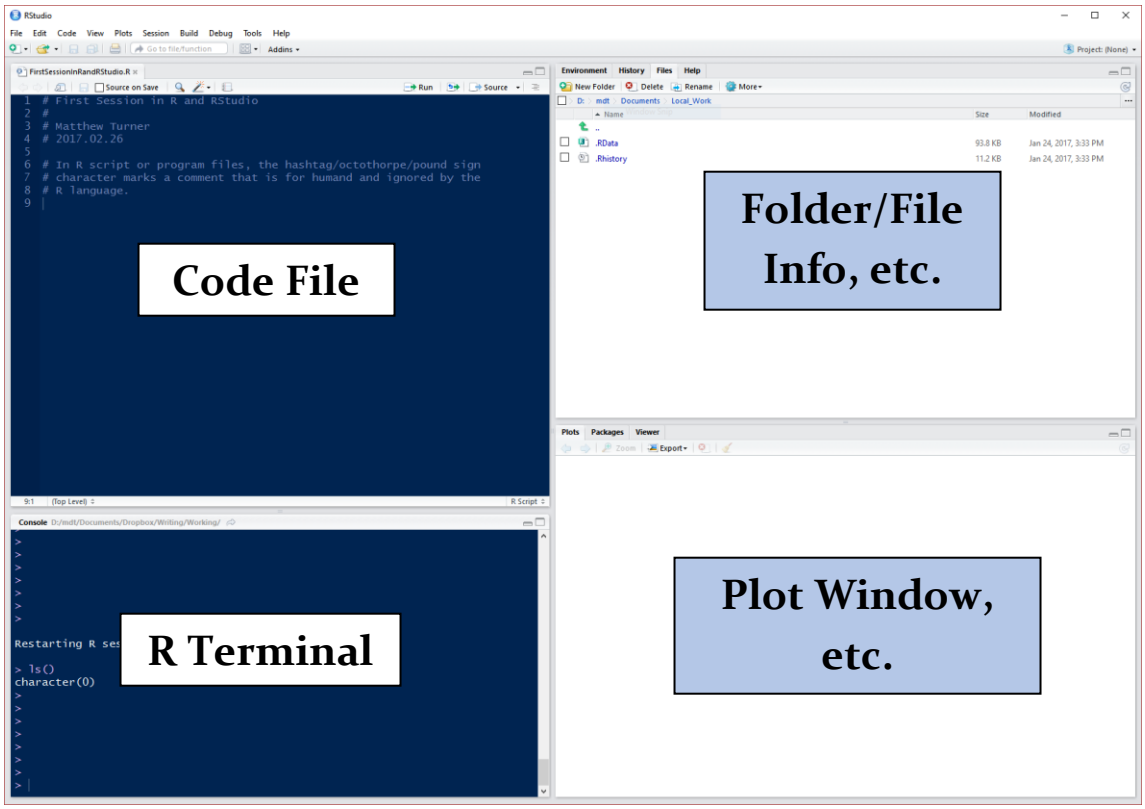
https://github.com/theEducationalCollective/SEPA2017-R-Workshop

and we welcome feedback: errors, typos, thank-you's (*it could happen!*), whatever, at the Github site.

SESSION

The RStudio Environment

The RStudio environment is a single window that is made up of 4 panes. Each of the 4 panes does different things. (Note that you can change where certain things appear in the display—see the "Global Options" on the "Tools" menu.) The main features of each pane are:



Additionally there are panes that allow you to load and look up information on R libraries, get help with R commands, see your files, see what variables you have created, and see what commands you have used recently.

A good workflow is to write your R commands (code) in the **Code File pane**, and send the code to the **R terminal pane** (where R runs each command) one line at a time.

This is done with the keystroke ctrl-ENTER (hold the control/ctrl key and press enter simultaneously) or, on the Mac the command-RETURN combination.

R as a Calculator

R can be used like a calculator by simply typing math into it, either in a code file or directly at the command line. Please see the provided code file for all of the examples in the workshop, the results (for part of the demonstration) look like:

```
>
> x = c(1,2,3,4,5)
> x
[1] 1 2 3 4 5
> mean(x)
[1] 3
> median(x)
[1] 3
> sd(x)
[1] 1.581139
> var(x)
[1] 2.5
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      1       2       3       3       4       5
>
```

Assignment Operator (<- versus =)

For historical reasons R has two assignment operators, <- (the left-pointing arrow, made up of the less-than sign and a dash) and = (the usual equals sign). In programming, **assignment**, that is, giving a name to something, is often distinguished from other uses of the equals sign. The <- operator is a reflection of this fact.

In most cases, <- and = are interchangeable. However, both are in use, so you need to recognize both as valid. The code file for this demonstration shows several examples (which work) and some that fail.

Note that the arrow can be "pointed" the other way (last assignment statement in the picture to the right) and while this is valid syntax for R, it is not recommended for use—it is considered "bad style." However, with the equals sign for assignment, **the name being given to something must always be on the left, and the "something" on the right.** It cannot be reversed!

```
> y = c(1,2,3,4,5)
> x <- c(5,4,3,2,1)
> s <- y + x
> y + x -> z
>
> x
[1] 5 4 3 2 1
> y
[1] 1 2 3 4 5
> s
[1] 6 6 6 6 6
> z
[1] 6 6 6 6 6
```

Installing and Attaching Libraries

The vast majority of R's functionality is provided via R Packages or Libraries. These are add-on code collections that implement additional statistical tests and procedures. As mentioned in the introduction this is good as it allows anyone to provide new tools for data analysis in R. Unfortunately, it also means that packages might vary in quality or usage as they are not controlled by a central authority. Usually this is not a problem.

To install a library, something you only need to do once for a given R installation, you use the **`install.packages()`** command in R. If you are online, and everything is working correctly it looks like this (to install the package named "**`e1071`**"):

```
>
> install.packages("e1071")

  There is a binary version available (and will be installed) but
  the source version is later:
      binary source
e1071  1.6-7  1.6-8

trying URL 'https://cran.rstudio.com/bin/windows/contrib/3.1/e1071_1.6-7.zip'
Content type 'application/zip' length 814561 bytes (795 KB)
opened URL
downloaded 795 KB

package 'e1071' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\mdt\AppData\Local\Temp\Rtmp4SLwZO\downloaded_packages
>
```

At the workshop, we will not be able to demonstrate installing packages from the internet as the conference is not able to provide internet access. In the future, when you are using R, adding new packages is very easy to do. The example here is e1071, which is a package for machine learning. We will not be using this package today, it was selected just for demonstration purposes.

Again—**once a package is installed, it does not need to be installed again**. This process permanently adds the package to the local R system.

To use a package, you load it with the **`library()`** command. This is an example of R's taciturn nature; when a package successfully loads, R basically says nothing about it. R **will** comment if something goes wrong.

```
>
> library(MASS)
>
>
```

The **`library`** command is needed every time you want to load a package for use in R. It is usually only required once per R session.

Data and Data Sets

R has many built in data sets, mostly from textbooks and other types of tutorials, as well as data from the published statistical literature. This can be very helpful for learning about statistical techniques. Here we load the "mammals" data set from the "MASS" R package. See the R code file for the commands and some more details.

The **`library`** command loads the MASS package, the **`data`** command tells R to go and look for the data set called "mammals." The data command will fail, if the MASS package is not loaded.

```
>
> data(mammals)
Warning message:
In data(mammals) : data set 'mammals' not found
>
> library(MASS)
> data(mammals)
>
```

Assuming that everything is done correctly, the data set is loaded, and we can use commands like **`head`** and **`summary`** to explore the data. The **`head`** command shows the column headings (variable names) and the first few rows of the data set, while the **`summary`** command gives a statistical summary of **each variable**:

```
> library(MASS)
> data(mammals)
> head(mammals)
                body brain
Arctic fox     3.385  44.5
Owl monkey     0.480  15.5
Mountain beaver 1.350   8.1
Cow          465.000 423.0
Grey wolf     36.330 119.5
Goat          27.660 115.0
> summary(mammals)
      body              brain
 Min.   :   0.005   Min.   :   0.14
 1st Qu.:   0.600   1st Qu.:   4.25
 Median :   3.342   Median :  17.25
 Mean   : 198.790   Mean   : 283.13
 3rd Qu.:  48.203   3rd Qu.: 166.00
 Max.   :6654.000   Max.   :5712.00
>
```

The summary provided by R is the so-called **five-number summary** of exploratory data analysis (EDA).[1] This is the **minimum, 1st quartile, median, 3rd quartile, and maximum**. Throughout R you will find many of the standard methods of modern statistics, such as the five-number summary in use as defaults. This is sensible as this software was written by

[1] Hoaglin, Mosteller, and Tukey (1983) *Understanding Robust and Exploratory Data Analysis*, Wiley. Although this book is older, it is still the bible of EDA. It is directly relevant to R, because the first version of R, called S, was invented at Bell Labs where many of the developers of EDA worked. The principles of EDA are hardwired deeply into the core of R.

statisticians for statisticians. Unfortunately, it means that you will often not see things you might be used to seeing in other software such as SPSS. For instance, the summary does not include means of standard deviations as the summary does not assume that the data are normal or otherwise suited to such measures.

> *Aside: Generic Functions*
>
> The summary command is the first example of what are called "generic functions" in R. These are functions that do different things when given different inputs. You will see the summary command used throughout the workshop today; it is used to summarize data sets, linear models, and other things. The command is smart—it provides the most relevant output for whatever you give it.

Data Frames

The mammals data set is the first example of what R calls a **data frame**. For the purposes of getting started with R, it is sufficient to think of data frames as **tables** or **spreadsheets**. (This will be covered in more detail later on in the workshop.) To see the variables that make up the data frame, you can use **`summary`** (as above) or the **`str`** command:

```
>
> str(mammals)
'data.frame':   62 obs. of  2 variables:
 $ body : num  3.38 0.48 1.35 465 36.33 ...
 $ brain: num  44.5 15.5 8.1 423 119.5 ...
>
>
```

The **`str`** command tells you what the object that you give it is (in this case a data frame), what its parts are (body and brain; the variables), some representative values, and a count of observations and variables.

Referring to Variables in Data Frames

Statistics such as the mean and standard deviation can be obtained with the commands shown above.

However, these commands need to refer to the variables directly. This is done with the object notation (also called the dollar-sign notation). The way to refer to a variable within a data frame is like this: **`dataframe$variable`** for the mammals data set it would be **`mammals$body`** (for the body weight variable).

```
>
> mean(mammals$body)
[1] 198.79
> sd(mammals$body)
[1] 899.158
>
```

For data frames that you intend to use a lot, you might **`attach`** them or use the **`with`** command. Both of these commands are in the code file for this demonstration.

Exploring Data

To demonstrate some of the above principles, we will now do some small analyses of data sets from within R. The first uses the eruption data from the Old Faithful geyser in Yellowstone National Park, Wyoming, USA. First we load and take a look at the data:

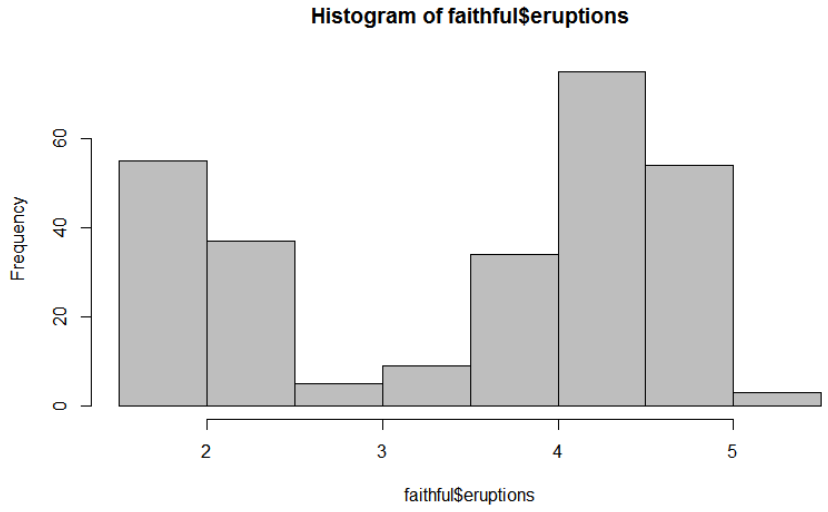
```
> data(faithful)      # The data is built-in
> str(faithful)
'data.frame':   272 obs. of  2 variables:
 $ eruptions: num  3.6 1.8 3.33 2.28 4.53 ...
 $ waiting  : num  79 54 74 62 85 55 88 85 51 85 ...
> summary(faithful)
   eruptions        waiting
 Min.   :1.600   Min.   :43.0
 1st Qu.:2.163   1st Qu.:58.0
 Median :4.000   Median :76.0
 Mean   :3.488   Mean   :70.9
 3rd Qu.:4.454   3rd Qu.:82.0
 Max.   :5.100   Max.   :96.0
>
```

The "eruptions" variable gives the length of eruptions (minutes), while the "waiting" variable gives the waiting time between eruptions (minutes).

While numerical summaries are good, **you should always make pictures to see the data!**

For this data set we can look at the distribution (histogram) for each variable, the correlation between the variables, and the scatterplot of their values. When working on analyses, R makes a number of graphs that are meant for work, but not for publication. Although by default R graphs are not usually publication quality, R has many systems in place to make high quality graphics for publications, posters, slides, etc.

The histogram for the eruption data is obtained by use of the command: **`hist(faithful$eruptions, col="gray")`** and the figure is below.

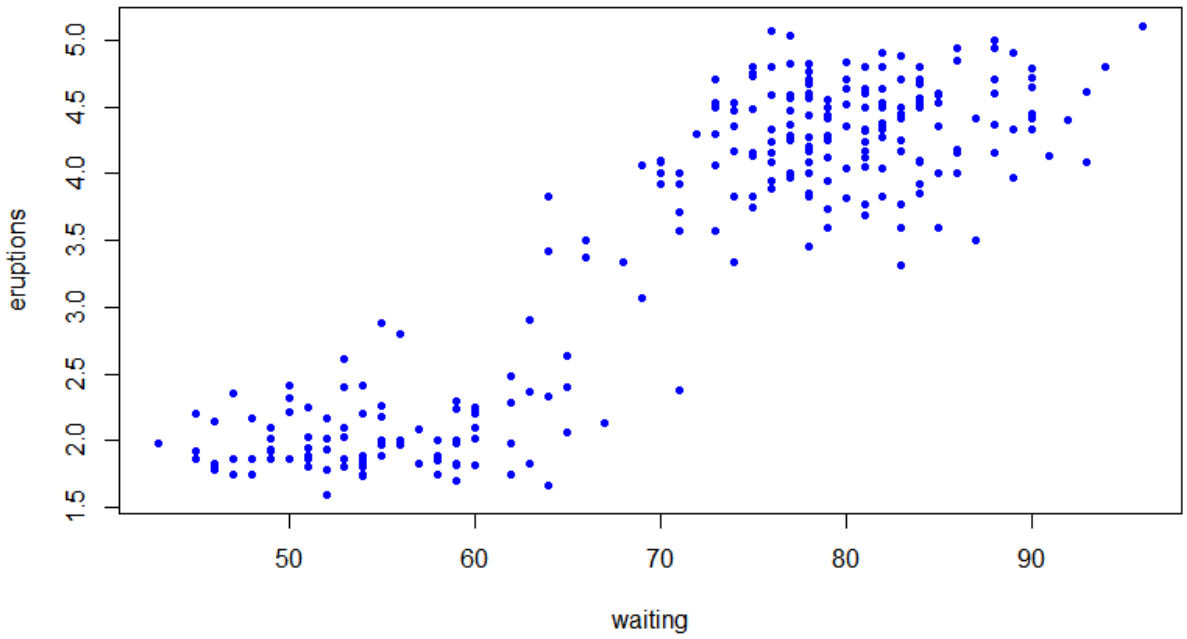


Note that by default the variable names are as they are specified in R, and the title is not very informative (or at least not useful for a publication). The **`col`** argument given to R tells the computer to **color** the bars "gray" which looks better in a document. Without it,

R defaults to bars in the histogram that are white with black lines. There are a wide variety of color options available.

We might also want to see the correlation between the variables:

```
with(faithful, plot(waiting, eruptions, pch=20, col="blue"))
```



The only new elements here are the use of **`plot`** rather than **`hist`**, I added the **`pch`** setting for plot that changes the plotting symbol, and changed the color to blue, just to show some differences.

Looking on Google will reveal many symbols available for use in R plots. Also, there are much more sophisticated plotting systems available in R via packages. Search the internet for "ggplot" to learn more!

We have a little more exploring to do but we have to digress for a moment and take a look at R's **model language** (also called **model *formulae***).

Important! The R Model Language

R has a well-developed language for expressing statistical models. It is a shorthand for the traditional statistical language. As an example, a regression model with 2 variables might be written:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \epsilon_i$$

in traditional math language, and it is written in R as:

```
y ~ x1 + x2
```

Note that in this format, you just specify the deterministic part of the model; you list the response variable to the left of the tilde ("**~**") and the explanatory or predictor variables to the right. The constant (for β_0) is assumed.[2]

Interactions are specified by the colon ("**:**"):

```
y ~ x1 + x2 + x1:x2
```

which is the model with terms **x1**, **x2**, and the interaction **x1:x2**. There are further shortcuts (such as the star or asterisk operator) which we leave to later.

Using this model language we can build regressions models easily. In the case of the geyser data, there are some issues to be addressed given the fact that both distributions are bimodal, but checking the correlation shows that the data is not too badly behaved.

```
> with(faithful, cor.test(waiting, eruptions))

        Pearson's product-moment correlation

data:  waiting and eruptions
t = 34.089, df = 270, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.8756964 0.9210652
sample estimates:
      cor
0.9008112

>
```

This correlation results give us the courage to try to build a simple regression model. This is where we need to use the R model language. The main function for building simple linear

[2] To eliminate the constant term in the model, you add in a term "**- 1**" which is, at least, intuitive if not mathematically correct.

models in R is **lm** (for "linear model"). This is used for fixed effects models of all types. The **lm** command takes a model specification and a name of a data set.

```
> faith.mod <- lm(eruptions ~ waiting, data=faithful)
>
> faith.mod

Call:
lm(formula = eruptions ~ waiting, data = faithful)

Coefficients:
(Intercept)      waiting
   -1.87402      0.07563

>
```

Note that R says nothing in response to fitting the model. Typing the name of the model tells us more: we get the model specification and the estimated regression coefficients, but that is all.

To get the usual regression information, use **summary** which is built to give sensible information when applied to many different objects.

```
> summary(faith.mod)

Call:
lm(formula = eruptions ~ waiting, data = faithful)

Residuals:
     Min       1Q   Median       3Q      Max
-1.29917 -0.37689  0.03508  0.34909  1.19329

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -1.874016   0.160143  -11.70   <2e-16 ***
waiting      0.075628   0.002219   34.09   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4965 on 270 degrees of freedom
Multiple R-squared:  0.8115,    Adjusted R-squared:  0.8108
F-statistic:  1162 on 1 and 270 DF,  p-value: < 2.2e-16

>
```

This result is probably most of what you were expecting. You get the summary of the residuals, the coefficients of the model, their corresponding t-values (Wald statistics), R^2,

adjusted R^2, the residual standard error, and the F statistic for the model. If you want the full model ANOVA table, this can be obtained with the **anova** function:

```
> anova(faith.mod)
Analysis of Variance Table

Response: eruptions
           Df  Sum Sq Mean Sq F value    Pr(>F)
waiting     1 286.478 286.478  1162.1 < 2.2e-16 ***
Residuals 270  66.562   0.247
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
>
```

For most psychology journals, the information you need on the F-test is in the summary result (the p-value, the F-statistic, and the degrees of freedom) but the table is there if you need it.

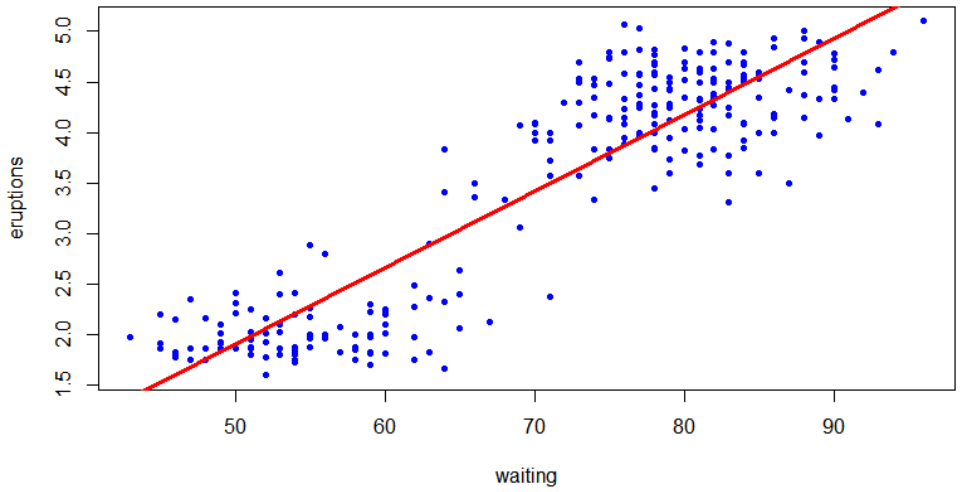
Finally, no regression would be complete without the line being plotted on the data. R is smart about a lot of standard analyses, so many commands know the reasonable thing to do when you feed them something. Specifically, the abline function knows how to draw many types of lines, and in particular the command:

```
abline(faith.mod, col="red", lwd=3)
```

will draw the regression line from the model faith.mod (in red with a line width of 3) on top of the last graph that was made. So the commands:

```
with(faithful, plot(waiting, eruptions, pch=20, col="blue"))
abline(faith.mod, col="red", lwd=3)
```

will produce:



Again, note that this model needs some additional work. While the regression line is reasonable for the data, the linear model's default handling of the variability depends on the data used being normally distributed and this is not (it is bimodal!) therefore this model badly predicts the values in the middle of the waiting range, between 65 and 75 minutes—specifically the model does not take into account the fact that very few values fall between 65-75 minutes of waiting time relative to the other times.

Next!

Most of the topics mentioned above will be addressed with further examples as the workshop proceeds.