An Introduction to R
Rough Draft*

The Berkeley R Project

UC Berkeley, Department of Statistics

VIGRE

John Jimenez†
The Berkeley R Project
johnjimenez@berkeley.edu

Devon Shurick‡
The Berkeley R Project
bigbird1013@gmail.com

September 8, 2008

*Though incomplete at this stage, the tutorial should get the beginner quite far.

†Sections 1.2 (examples from prof. Rice), 1.3, 2.2 (ending "—>" section), 2.3 (strsplit section), 2.6 (from accessing (i,j) elements on), 2.7, 2.8, 2.10.1, 3, 4, 5, 6, 8, and 9.

‡Sections 1.1, 2.1 - 2.3, 2.4 - 2.11, and 7

Contents

List of Figures

1 Introduction

1.1 What is R?

R is both a programming language and a software program bundled into one neat little package. Normally, software and languages are developed separately, and the program uses the language by compiling the language into something the software can understand, then finally running the program. R, however, is an interpreted language, which means that every command you type into the prompt is immediately read by the software and interpreted, without having to compile and build a whole program. Therefore, most of the things you do in R will be by typing one line at a time, at the command prompt, which is represented by the '`> `' symbol.

Another thing you must know is that R is composed of *objects*. These include functions, variables, data, etc., and are stored in the memory of the computer for later use. To perform action on these objects, we have *functions* and *operators*.

1.2 What can I do with R?

R is quite versatile, with capabilities ranging from data analysis to data scraping. We will explore some analyses that may be performed in R later, but for now here is a list of fundamental things R can do:
R is a calculator:

```
> (2+3)
[1] 5
> 2^3
[1] 8
> cos(4.7)
[1] -0.01238866
```

R can operate on scalar variables:

```
> x = 6
> 2*x
[1] 12
> exp(x)
[1] 403.4288
```

R can operate on vectors:

```
> x = c(1,2,3,4)
> x
[1] 1 2 3 4
> x[2]
[1] 2
> x+1
[1] 2 3 4 5
> x^2
[1]  1  4  9 16
> cos(x)
[1]  0.5403023 -0.4161468 -0.9899925 -0.6536436
> y = x + 3
> y
[1] 4 5 6 7
```

```
> x/y
[1] 0.2500000 0.4000000 0.5000000 0.5714286
```

R can do logical operations:

```
> x > 2
[1] FALSE FALSE  TRUE  TRUE
```

R can calculate statistics:

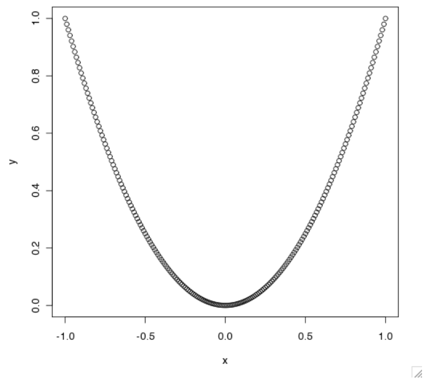
```
> mean(x)
[1] 2.5
> sd(x)
[1] 1.290994
```

R can plot:

```
> x = seq(from = -1, to = 1, by = .01)
> y = x^2
> plot(x,y)
```

Here, the `seq` is a function that creates a sequence, hence the name, of numbers `from` negative one `to` one, with each number separated `by` .01. If you're thinking this is very self-explanatory, it is. If you're thinking the typing is tedious, don't worry, `seq(-1, 1, .01)` does the same, but we'll get to that later.

Figure 1: What R graphs as a result of `plot(x,y)`:



R can generate random numbers:

```
> x = rnorm(1000,10,20)
> mean(x)
[1] 9.998576
> sd(x)
[1] 19.93155
> min(x)
[1] -50.95176
> max(x)
[1] 65.02984
> hist(x)
```

1.3 How Can I Get a Copy?

To obtain a copy of *R*, visit the site http://cran.cnr.berkeley.edu/, and visit one of the three links `Linux`, `MacOS X`, or `Windows` in the section `Download and Install R`. Then simply choose the distribution of *R* you desire.

1.4 The Focus of this Tutorial, and other References

This tutorial is written for people who have no experience with *R*. In turn, we only cover what we feel to be the most fundamental areas. This allows us to explain the fundamentals in more detail than might be found in more broadly focused tutorials. Detailed statistical examples have been provided wherever possible to show how to combine the topics covered. We strongly reccomended observing the help options for each function covered, to see their full capabilites; something not covered in this text. Such investigation will also lead to alternatives that may better suit the readers programming style.

This being said, for a more broad coverage of *R*, see An Introduction to R, by W.N. Venables, D.M. Smith and the *R* Development Core Team, at `http://www.r-project.org/`. Another is R for Beginners, by Emannual Paradis. Paradis explains in more detail the inner workings of *R*, with nice drawings, for those interested. For a summary of commands, vist the website `http://www.stat.berkeley.edu/ epurdom/RNotes.pdf`. Charlotte Wickham's Introduction to R contains simulation, with exercises and solutions, located at `http://cwick.co.nz/camp.html`.

2 Objects

We have said before that everything in R is an object. In order to better differentiate these objects, every object has a *mode* and a *length*. The *mode* gives the basic type of the elements of an object, and the four main modes are as follows:

- Numeric - A number; either an integer or a double (fraction).
- Character - A string or word.
- Logical - A TRUE or FALSE value.
- Complex - A complex number (i)

The *length* is the length of the object, or how many elements are contained within the object. You can find out the *mode* and *length* of any object using the `mode` and `length` functions.

```
> num1 = 3
> mode(num1)
[1] "numeric"
> char1 = "hello"
> mode(char1)
[1] "character"
> bool1 = TRUE
> mode(bool1)
[1] "logical"
> comp1 = 1i
> mode(comp1)
[1] "complex"
> length(num1)
[1] 1
```

```
> length(c(1, 2, 3, 4, 5))
[1] 5
```

Also, for all modes, missing values are always represented as `NA` (Not Available).

While working in *R* we will sometimes be dealing with a single number value, or sometimes even a large dataset. We need a way to store these objects or values for later use. That's where *variables* come in. Variables can be thought of as an attribute which may change its value while it is under observation. We usually give variables a name or a letter, in order to recognize that it is a variable. We have already seen an example of using variables when we wrote `char1 = "hello"` above. When saving a value to a variable, we call it an "assignment". Assignments take the result of the statement on the right of the '`=`' symbol and stores it in a variable whose name is given on the left. In place of the '`=`' symbol, we can also use the '`<-`' symbol. In the examples provided above, the result of the expression on the right is simply the number that we happened to type it. We then printed out a variable's value by typing the name of the variable.

Further note that just as `<-` says "Take what is on the right of this operator and store it into the name listed to the left of the operator", the symbol `->` says just the opposite. The intuition is the same in that we are storing into the direction of the arrow. The latter comes in handy when you have made a long computation and want to go back and store the value of that computation. For example, suppose you were calculating your estimated grade in a class as

```
> 90*.1 + 89*.15 + 91*.15 + 95*.20 + 91*.4
[1] 91.4
```

But if we wanted to store this, we could either re-enter the operation, with a `grade =` or `grade <-` at the beginning, or push the up arrow key to retrieve the previous line. If in a GUI, we could move the mouse cursor to the beginning of the line and enter the same `grade =` or `grade <-`. The easiest way of course is to push up and then `-> grade`. Though the example may seem trivial, keep this option in mind, because it will come in handy as your experience with *R* expands.

2.1 Numbers

Numbers make up the *numeric* mode. Numbers can be a simple integer

```
> a = 3
> a
[1] 3
```

They can be a decimal

```
> a = 3.50
> a
[1] 3.5
```

or they can be a large value in exponential notation.

```
> a = 3.0e24
> a
[1] 3e+24
```

They also have some special values, `Inf` (∞), `-Inf` ($-\infty$), and `NaN` (Not a Number).

```
> infty = 1/0
> infty
[1] Inf
```

```
> neg.infty = -1/0
> neg.infty
[1] -Inf
> infty + neg.infty
[1] NaN
```

We can also perform mathematical functions on numbers and variables:

```
> a = 3
```

$a(4)$

```
> a(4) #Attempts to pass the integer 4 to a function a, but it doesn't exist
Error: could not find function "a"
```

$\sqrt{a}$

```
> sqrt(a)
[1] 1.732051
```

a^5

```
> a^5
[1] 243
> a**5
[1] 243
```

$|-a|$

```
> abs(-a)
[1] 3
```

$a * 3/5 + 7 - 2$

```
> a*3/5+7/2
[1] 5.3
```

2.2 Vectors

Vectors are a variable in the commonly understood meaning: a listing of elements in one dimension that are indexed so that individual items can be selected later by one or more indices. In *R*, all the elements of a vector must be of the same *mode.* For a vector object, the *length* becomes the number of elements in the vector.

2.2.1 Creating Vectors

Vectors are most commonly created by using the 'c' function:

```
> vec = c(1, 8, 4, 2, 6)
> vec
[1] 1 8 4 2 6
> c(TRUE, FALSE, TRUE)
[1]  TRUE FALSE  TRUE
> c("hello", "world")
[1] "hello" "world"
```

When looking at a vector when printed out onto the window, the numbers in brackets (in this case the '[1]') tells you the index in the list of results of the element immediately following the bracketed number. So, since when the vector was printed out it only needed to use one line for output, you will see a '[1]' to begin the line of output; this means that the element immediately following the '[1]' is the first element in the vector. If there were enough numbers to use up more than one line of output, there will be one bracketed number per line of output, like so:

```
> numeric.vector = 1:50
> numeric.vector
[1]   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
[30] 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

Notice that the number '30' has a '[30]' in front of it. That is because it is the first element of that line of output and it is element number 30 in the vector. This numbering system becomes a little more complicated when dealing with lists, because we can have lists contained within lists. But you don't need to worry about that for now.

There are three other tools for vector creation that come in handy:

```
> seq(from = 1.575, to = 2.075, by = 0.05)
 [1] 1.575 1.625 1.675 1.725 1.775 1.825 1.875 1.925 1.975 2.025 2.075
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> rep(1:3, times = 3)
[1] 1 2 3 1 2 3 1 2 3
```

`seq` is a function that generates a sequence of numbers, beginning at `from` and ending at `to`, with the interval given by `by`. The next example, `1:10`, is much like a simplified version of the `seq` function, where it sequences automatically by from the first number to the second number, either by +1 or -1, depending on whether the first number is greater than the second. `rep` is a function that repeats a vector a designated number of times.

If you try to put objects of different modes into a vector, *R* will convert all elements to a mode which all the elements can be converted to:

```
> c(3, "three")
[1] "3"     "three"
> c(3, TRUE)
[1] 3 1
> c(3, 3i)
[1] 3+0i 0+3i
> c(3, 3i, "three")
[1] "3"     "0+3i"  "three"
> c(3, 3i, FALSE)
[1] 3+0i 0+3i 0+0i
```

We can also append two vectors together using the same `c` function:

```
> vec2 = c(5, 3, 7)
> vec3 = c(vec, vec2)
> vec3
[1] 1 8 4 2 6 5 3 7
```

2.2.2 Indexing Vectors

Now we can index one or more of its elements by using brackets containing the indeces we want following the name of the vector:

```
> vec
[1] 1 8 4 2 6
> vec[2]
[1] 8
```

If we want to grab more than one element, we can provide another vector inside the brackets, with an element for each index we wish to grab:

```
> vec[c(1,2)]
[1] 1 8
> vec[c(1,1,3)]
[1] 1 1 4
```

If we pass a logical test on the vector, out pops a logical vector telling which elements of the vector pass that test:

```
> a = 1:10
> a > 6
 [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

Now we can use this logical vector to subset the original by putting the logical test within brackets like so:

```
> a[a > 6]
[1]  7  8  9 10
```

Notice how it only returns the values of `a` that are greater than 6. Here is one more example:

```
> a != 3
 [1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> a[a != 3]
[1]  1  2  4  5  6  7  8  9 10
```

2.2.3 Vector Arithmetic

Similarly to numbers, we can also do arithmetic on vectors:

```
> a = 1:3
> b = 4:6
> a+b
[1] 5 7 9
> a*b
[1]  4 10 18
> a^2
[1] 1 4 9
> a/b
[1] 0.25 0.40 0.50
```

2.3 Strings

Anything in between pairs of single or double quotes are defined as members of the *character* class, also called a "string":

```
> b <- "Category A"
```

We can convert numeric objects to character objects and back:

```
> pi = 3.14
> pi = as.character(pi)
> pi
[1] "3.14"
> pi = as.numeric(pi)
> pi
[1] 3.14
```

We can also paste strings together:

```
> a = "string1 +"
> b = "string2"
> paste(a,b)
[1] "string1 + string2"
```

and use them in vectors just like numeric objects:

```
> x = "element 1"
> y = "element 2"
> c(x,y)
[1] "element 1" "element 2"
```

To split a string, we use the `strsplit` function, with the form `strsplit(string to be split, what to split the string by)`. For example, to split the string "1234567890" by each empty string "", do the following :

```
> strsplit("1234567890", "")->myString; myString
[[1]]
 [1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "0"
```

Note the double brackets around the 1 above, in `[[1]]`. This states that `myString` is a `list`. The `[[1]]` denotes that what follows is the first element in the list. To refer to this element, we must call `myString[[1]]`. We will put this to use in a moment. Of course, it always nice to know how to put things back together once we have taken them apart. This can be achieved here with the `paste` function, with the form `paste(what to paste together, what you want to separate these items with)` . For example, if we wanted to put the string back to it's original form, we would paste the elements of `myString`, separating each element with an empty character :

```
paste(myString[[1]], collapse = "")
[1] "1234567890"
```

We will cover other String manipulations and issues related to data frames in the the sections on reading data into R[1].

[1] If you just can't wait, check out `gsub` with the call `help(gsub)`.

2.4 Factors

A `factor` is a categorical variable, that can be of either the `numeric` or `character` mode. They can be used when needing to categorize data into different groups. A factor includes a vector of 'labels' for the categories, as well as number of different levels that the factor contains. Factors can be created using the `factor` function:

```
factor(x, levels = sort(unique.default(x), na.last = TRUE),
     labels = levels, exclude = NA, ordered = is.ordered(x))
```

The first argument, `x`, is a vector of data, that will be attempted to convert into a `factor` object.

```
> factor(1:5)
[1] 1 2 3 4 5
Levels: 1 2 3 4 5
> factor(c("high", "high", "low", "medium"))
[1] high   high   low    medium
Levels: high low medium
```

Notice how the `levels` contain only the unique values found in the vector of values. The `levels` argument designates all the possible levels of the factor, which includes all possible values that the data could have taken on. Notice that by default this is set to be all of the unique values of the vector `x`. Here is an example where we specify the levels ourselves:

```
> factor(1:3, levels = 1:7)
[1] 1 2 3
Levels: 1 2 3 4 5 6 7
```

Notice that all of the values in `x` were also found in the `levels` vector we specified. However, if values in `x` are not found within the vector of `levels`, they are replaced by `<NA>` values:

```
> factor(1:3, levels = c("A", "B", "C", "D", "E"))
[1] <NA> <NA> <NA>
Levels: A B C D E
> factor(1:3, levels = c("A", 3, "C", 2, 1))
[1] 1 2 3
Levels: A 3 C 2 1
```

`labels` determines the names of the levels. If you decide to specify the labels of the factor, the vector of labels must be of the same length as the number of different levels, and R automatically determines which label goes with which value by assigning the next unique value in the vector `x` with the next unique label in `labels`. Here is an example where we specify the labels ourselves:

```
> factor(1:3, labels = c("low", "medium", "high"))
[1] low    medium high
Levels: low medium high
```

The number of labels must equal the number of levels, which again is the number of unique values found in `x`.

```
> factor(c(2, 3, 3, 1, 2), labels = c("low", "medium", "high"))
[1] medium high   high   low    medium
Levels: low medium high
```

Notice that every ‘1’ is replaced by ’low’, ‘2’ replaced by ’medium’, and ‘3’ replaced by ‘high’. This is because ‘1’ is the first of the ordered unique values, and ‘low’ is the first label found in `labels`. `exclude` is a vector of values to be excluded when forming the set of levels:

```
> factor(1:3, exclude = 2)
[1] 1    <NA> 3
Levels: 1 3
> factor(1:3, levels = c("A", 3, "C", 2, 1), exclude = 2)
[1] 1    <NA> 3
Levels: A 3 C 1
```

Notice how if a value in `exclude` is found within the vector of values, then it is replaced by `<NA>` and removed from the `levels` as well. If you set `exclude` to be `null`, the missing value (NA) is treated as a valid level, as in:

```
> factor(c(3, 3, 2, 8, 6, 4, 2, NA), exclude = NULL)
[1] 3    3    2    8    6    4    2    <NA>
Levels: 2 3 4 6 8 <NA>
```

`ordered` is a logical argument used to specify whether the levels should be regarded as ordered. The default value is determined by `is.ordered(x)`, which tells if the values in `x` are ordered or not.

```
 factor(c("high","high","low","medium"),
+ levels=c("low","medium","high"),ordered=FALSE)
[1] high   high   low    medium
Levels: high low medium
 factor(c("high","high","low","medium"),
+ levels=c("low","medium","high"),ordered=TRUE)
[1] high   high   low    medium
Levels: high < low < medium
```

Notice that if `ordered` is set to `TRUE`, ‘high’ is considered earlier in order than ‘low’ or ‘medium’.

2.5 List

A list is pretty much what sounds like: a listing of objects. Lists can contain any object, even lists. In fact, when lists are created, every object within the list is converted to a list object. Therefore, lists can go several layers deep:

```
> list(x, y)
[[1]]
[1] 1 2 3 4

[[2]]
[1] 1 2 3 4 5 6 7 8

> list(x, list(y))
[[1]]
[1] 1 2 3 4

[[2]]
[[2]][[1]]
[1] 1 2 3 4 5 6 7 8
```

Notice that the indexing style for lists is different. `[[1]]` is indicating that this is the first list in the list, and `[1]` is indicating the first element of the vector contained in that list. We can then select either the list objects or the objects they contain, depending on the sets of brackets we use:

```
> l1 = list(x, y)
> l1
[[1]]
[1] 1 2 3 4

[[2]]
[1] 1 2 3 4 5 6 7 8

> l1[1]
[[1]]
[1] 1 2 3 4

> l1[[1]]
[1] 1 2 3 4
```

`l1[1]` grabs the first list in the list, whereas `l1[[1]]` grabs the object contained in that list. Like data frames, the elements of a list can also be named.

```
> l2 = list(independent = x, dependent = y)
> l2
$independent
 [1]  3.03  5.53  5.60  9.30  9.92 12.51 12.95 15.21 16.04 16.84

$dependent
 [1]  3.19  4.26  4.47  4.53  4.67  4.69 12.78  6.79  9.37 12.75
```

We can now access these elements using the '$' notation:

```
> l2$dependent
 [1]  3.19  4.26  4.47  4.53  4.67  4.69 12.78  6.79  9.37 12.75
```

2.6 Matrices

A matrix is just a vector in 2 dimensions, so therefore it has a vector of values, and a `dim` attribute which specifies the number of rows and columns of the matrix. To see some examples, we will be creating matrices using the `matrix` command.

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

`data` is the vector of data to be used. `nrow` and `ncol` specify the number of rows and columns, respectively. `dimnames` can be used to specify the names of the rows and columns, by passing a list of length 2, containing a vector of names for the rows and a vector of names for the columns.

```
> matrix()
     [,1]
[1,]   NA

> matrix(1:6)
     [,1]
```

```
[1,]    1
[2,]    2
[3,]    3
[4,]    4
[5,]    5
[6,]    6
> matrix(1:6, ncol = 2)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> matrix(1:6, ncol=4)
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    1
[2,]    2    4    6    2
Warning message:
In matrix(1:6, ncol = 4) :
  data length [6] is not a sub-multiple or multiple of the number of columns [4]
```

Notice that when specifying the number of rows or columns, if the length of the data is not a multiple of the rows, columns or rows and columns, then the function wraps the data until it fills the necessary dimensions. `byrow` is a logical option telling the function to fill the data along the rows. By default, it is set to `FALSE`, so the function fills down the columns.

```
> matrix(1:6, ncol = 2, byrow = TRUE)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Given that a matrix is two dimensional, we can call on its elements by row and column. Suppose we had a matrix as follows :

```
> matrix(1:6, ncol = 2, byrow = TRUE)->DATA
```

Now to access the i^{th} row and j^{th} column, we use the form `DATA[i,j]` to obtain that element. For example,

```
> DATA[1,2]
[1] 2
```

Although matrices are usually indexed with two subscripts, it's still valid to use just one, in which case the matrix is treated like a vector consisting of the columns of the matrix. In other words, the matrix is referenced like a vector, whose elements are ordered by the columns. For an example:

```
> DATA[5]
[1] 4
```

To expand a matrix by row or column, we use the `rbind or cbind` functions respectively with the form `rbind{data to bind to, data to bind}`. For example, we could bind the vector (1, 2, 3), as a column and then as a row, to `DATA` as follows :

```
> cbind(DATA, 1:3)->DATA; DATA
     [,1] [,2] [,3]
[1,]    1    2    1
[2,]    3    4    2
[3,]    5    6    3
> rbind(DATA, 1:3)->DATA; DATA
     [,1] [,2] [,3]
[1,]    1    2    1
[2,]    3    4    2
[3,]    5    6    3
[4,]    1    2    3
```

To remove a row or column, use the form `DATA[-i,]` or `DATA[, -j]` to remove the i^{th} or j^{th} column respectively. To get back to our original matrix DATA :

```
> DATA = DATA[-4,]; DATA = DATA[,-3]; DATA
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

The same rules apply with removing multiple columns or rows at once with the colon, `:`, operator.

If we think of `DATA[i,j]` as saying "I want DATA, such that I am in the i^{th} row and j^{th} column" we can begin to see new ways of accessing data. For example, `DATA[DATA[,2]==0,]` says "I want DATA, such that I am in a row where the second column of DATA is equal to zero." This can come in handy when having to subset your data (try `help(subset)` for more on subsetting). The initial step is to decide what we want, in this case rows, conditional on the columns meeting some requirement. Note, the setup positions the logical requirement in the row position, stating we want rows. The logical condition simply states which rows we want. For example,

```
> DATA[DATA[,2] == 2, ]
[1] 1 2
> DATA[DATA[,1] == 5, ]
[1] 5 6
```

This should get you quite far in your subsetting tasks.

2.7 Data-Frames

While a matrix can contain only one type of data, a data-frame may hold many different types, so long as individual columns of the data-frame contain only one type throughout that column. To create a data-frame, we use the `data.frame` function, with one of many forms. One way is to create a data-frame from vectors : `data.frame(v1, v2, v3, ..., vn)`, where each `v` is a vector. In this case the columns of the data-frame will take on the names of the vectors. Another way is to create a data-frame from a matrix or vector : `as.data.frame(m1)`, where `m1` is a matrix or vector. If you want to set the names of the columns on your own, use the function `names` with the form `names(your data-frame here) = c(c_name1, c_name2, ..., c_namen)`, where each c_namei is the name you want to give to column i. Once a data-frame is created you may add a column or row to it using basic indexing. However, with data-frames, we have many ways of refrencing our variables (columns). For example, suppose we created the following data-frame :

```
> v1 = 1:10
> v2 = 2:11
> v3 = 11:2
> d1 = data.frame(v1, v2, v3)
> d1
   v1 v2 v3
1   1  2 11
2   2  3 10
3   3  4  9
4   4  5  8
5   5  6  7
6   6  7  6
7   7  8  5
8   8  9  4
9   9 10  3
10 10 11  2
> names(d1) = c("col1", "col2", "col3")
> d1
   col1 col2 col3
1     1    2   11
2     2    3   10
3     3    4    9
4     4    5    8
5     5    6    7
6     6    7    6
7     7    8    5
8     8    9    4
9     9   10    3
10   10   11    2
```

Now, `d1[,2]` is equivalent to `d1$col1`, which is equivalent to `d1[,'col1']`. That is, instead of saying we want `d1` such that we are in the second column, if it is easier for us to remember the name of the second column, we may use that instead, by placing a $ after the data-frame name and before the column name, or call the data-frame column, refrencing the column name instead of its location. For example, the following asks whether or not the first sentance of this paragraph is true, and the notation will be covered soon.

```
> d1$col1 == d1[,1]; d1[,1] == d1[,'col1']
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

More on this in a moment. The following code displays one of the many ways to add a column to `d1`.

```
> d1[,4] = 21:30; names(d1)[4] = "col4"
> d1
   col1 col2 col3 col4
1     1    2   11   21
2     2    3   10   22
3     3    4    9   23
4     4    5    8   24
5     5    6    7   25
```

```
6      6     7    6   26
7      7     8    5   27
8      8     9    4   28
9      9    10    3   29
10    10    11    2   30
> d1[,5] = strsplit("1234567890", ""); names(d1)[5] = "String Column"
> d1
   col1 col2 col3 col4 String Column
1     1    2   11   21             1
2     2    3   10   22             2
3     3    4    9   23             3
4     4    5    8   24             4
5     5    6    7   25             5
6     6    7    6   26             6
7     7    8    5   27             7
8     8    9    4   28             8
9     9   10    3   29             9
10   10   11    2   30             0
```

We can delete columns just as we can with matrices to obtain our original data-frame, by using the negative subscript as follows

```
> d1 = d1[,-4:-5]
> d1
   col1 col2 col3
1     1    2   11
2     2    3   10
3     3    4    9
4     4    5    8
5     5    6    7
6     6    7    6
7     7    8    5
8     8    9    4
9     9   10    3
10   10   11    2
```

Just as a Matrix, with the name `MatrixName`, can be subsetted with a call like `MatrixName[MatrixName[,j] == 1, ]`, to call on some matrix `MatrixName` where its j^{th} column equals one, so too can a data-frame be referenced. However, with the data-frame, say `dataName` we could equally make use of the fact that data-frame columns have names, and do the following. Supposing a column name of a data-frame `dataName` was `colName`, we could use `dataName[ dataName$colName == 1, ]`, instead of using the more cryptic `dataName[ dataName[,j] == 1,]`. Similar use with `dataName[,'colName']` works[2].

2.8 Tables

The `table` function will create a table counting the number of occurances of a factor in what object is passed to it. For an illustrative example, we prematurely introduce you to the `rnorm` function, which has the form `rnorm(how many numbers to generate, mean, sd)` . So, to create and store in, say `OurRV`, 1000 randomly generated observances of a random variable following a Normal distribution

[2]Some functions you may be interested in at this point, with respect to data-frames are `subset`, `with`, `head`, `tail`, and `summary`

with mean 0 and standard deviation 3, we would write `OurRV = rnorm(1000, 0, 3)`. The `round` function is also used in the following example, with the form `round(number to round, by how many decimal places)`, so `round(3.5, 0)` results in the integer 3. To observe through a table the distribution of counts of randomly generated $N(0, 9)$ observances, when rounded to integers, we could do the following :

```
> table(round(rnorm(1000, 0, 3), 0 ) )

 -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4   5   6   7   8   9  10
  2   3  11  22  40  65  66  86 134 115 134 103  70  66  39  20  13   6   2   3
```

We may also compare two objects to eachother. For example, we could see how closely generating two sets of $N(0, 9)$ variables match eachother, when rounded to the nearest integer, element by element; that is, if the i^{th} element of the first vector matches the j^{th} element of the second vector, then `table` will represent this with a count.

```
> table(round(rnorm(1000, 0, 3), 0), round(rnorm(1000, 0, 3), 0))

      -10 -9 -8 -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7  8  9 12
  -12   0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0
  -8    0  0  0  0  0  0  0  0  0  0  0  0  0  0  2  1  0  0  0  0  0
  -7    0  0  1  0  0  0  0  0  0  2  1  0  0  2  0  0  0  0  0  0  0
  -6    0  0  0  0  0  0  1  2  1  2  2  1  3  3  0  0  1  0  0  0  0
  -5    0  0  0  0  0  3  2  3  2  3  3  1  0  3  1  0  1  0  0  0  0
  -4    0  0  0  0  1  0  3  5  6  7 11  4  9  5  3  2  0  1  0  0  0
  -3    0  0  0  0  2  1  3  2 13 13 14 13 11  4  6  7  1  1  1  0  0
  -2    0  0  1  0  0  1  3 17 10 14 13 12  8  9  5  4  1  1  1  1  0
  -1    0  1  0  0  3  6  9  3 13 14  8 15 11 12  5  4  0  2  1  1  0
  0     0  0  1  3  2  4  5 14 15 26 10 14 14  6 12  4  0  2  1  0  0
  1     1  0  0  1  2  4 11 13  9 21 19 13 15 15 10  6  2  1  1  0  0
  2     0  0  0  1  2  1 11  7 10 11 16 11 10  7  3  3  3  0  0  1  0
  3     0  0  0  1  2  0  4  8 13  8 11 15 11  9  7  2  0  1  0  0  0
  4     0  0  0  2  1  1  5  3  8 10 10  2  4  5  3  0  3  0  1  0  1
  5     0  0  0  1  1  1  3  4  3  2  4  5  1  2  3  1  2  0  0  1  0
  6     0  0  0  0  0  1  1  2  4  1  1  5  2  1  2  0  0  0  0  0  0
  7     0  0  0  0  0  0  1  1  0  0  1  2  0  1  1  0  1  0  0  0  0
  8     0  0  0  0  0  0  0  0  1  1  0  1  0  1  0  2  0  0  0  0  0
  9     0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0
```

This is called a contingency table. A better example might compare, say, the number of days it takes different plants to grow to a certain height under different treatments.

2.9 Operators

Operators are a special type of function. They are functions that are used more often, so the designers decided to make it more easy to use them[3]. They do not use the '*name*(*arguments*, *operators*, ...) format, so they lose some of the control that functions provide. Operators include things such as arithmetic (`+`,`-`, `*`, ...), comparison (`<`, `>`, `<=`, `>=`), and logical (`==`, `!=`, `&&`, `||`). As you can see, it is much easier to do something like

[3]Some more information on the operators that R provides can be found here: http://www.statmethods.net/management/operators.html

```
> 2+2
```

than it is to do

```
> add(2, 2)         # Won't work
```

which won't even work unless you create a function called `add = function(x,y) = x + y`, though such would not make much sense, since `+` works fine for adding two numbers.

2.10 Other Things You Must Know About R

- **R is Case Sensitive** - When you're dealing with names of variables, functions, etc., be aware that R is case sensitive, so "item1" is a different variable than "Item1".
- **R Ignores whitespace** - The only time that whitespace becomes important is when you're creating names for variable and functions: they must contain no form of whitespace.

  ```
  > item one = 3
  Error: unexpected symbol in "item one"
  ```

  Other than this, the R interpreter ignores all whitespace (spaces, tabs).
- **Commands Are Separated by Either a ";" or a Newline** - You can either type

  ```
  > 2+2
  [1] 4
  > 3+3
  [1] 6
  ```

  or

  ```
  > 2+2;3+3
  [1] 4
  [1] 6
  ```

  If you do not complete a command before hitting 'enter', the prompt will continue to the next line with a '+' prompt, allowing you to continue typing in the command on the next line.

  ```
  > mean(c(1, 2, 3, 4),
  + na.rm = TRUE)
  [1] 2.5
  ```

- **Sessions** - As you know, R saves all objects in memory When you load R for the first time, a brand new session is created and no new obects (those other than *R*'s defaults) have been created. At any time during your session, you can save your session by selecting `File > Save Workspace`. Also, when you exit the R interface, you will be prompted as to whether you would like to save your session. The next time you load R, all of the objects that you created during your last session will be restored.
- **rm() and ls() commands** - To view all of the objects that you have created so far, use the '`ls()`' command; this will list the names of any variables or functions that you have created and are currently stored in memory. To remove all of these objects, simply use the '`rm()`' command, like so:

```
> num1 = 3
> ls()
[1] "num1"
> rm("num1")
> ls()
character(0)
```

- **Up/Down Arrow Keys Cycle Through History** - All of your previous commands are remembered in your session's history. To cycle through them, you may use the up and down arrow keys, the up arrow key giving you the previous command and the down arrow giving you the next command. If you entered in a command across multiple lines of prompt, the history will save the command line-by-line as well.
- **Tab Completion** - Sometimes, after you have created lots of objects, you might begin to run out of short, creative names for them. That's where tab completion comes in handy. Tab completion makes it faster to reference a variable or function name. After typing in n characters if you press the 'tab' key once, sometimes the prompt will automatically choose an object name that it thinks you are trying to obtain and fills in the rest of the name automatically. If this is not the object name that you were looking for, you can press the 'tab' key until you get a list of all objects that have a name that begins with those n characters. For example, if you type in

  ```
  > mea
  ```

  and then press the 'tab' key **once**, R thinks that you are trying to get the 'mean' command, so it will fill in the rest of the command like so:

  ```
  > mean
  ```

  If this is not the object name that you were looking for, you can press the 'tab' key until you obtain a list like so:

  ```
  > mean
  mean            mean.data.frame mean.Date       mean.default    mean.difftime
  mean.POSIXct    mean.POSIXlt
  ```

2.10.1 Logical and Comparison Operators

Logical and comparison operators result in an expression being true or false. For example :

Question : First, is 3 equal to 4? Secondly, is 3 not equal to 4?

```
> 3 == 4; 3 != 4
[1] FALSE
[1] TRUE
```

Question : First, is 3 less than 4? Secondly, is 3 less than or equal to 4?

```
> 3 < 4; 3 <= 4
[1] TRUE
[1] TRUE
```

Question : First, is 3 greater than 4? Secondly, is 3 greater than equal than 4?

```
> 3 > 4; 3 >= 4
[1] FALSE
[1] FALSE
```

Question : First, is 3 greater than 4 and (`&&`) 4 greater than 3? Secondly, is 3 greater than 4 or (`||`) 4 greater than 3?

```
> 3 > 4 && 4 > 3; 3 > 4 || 4 > 3
[1] FALSE
[1] TRUE
```

You will find that these come in handy when subsetting data and programming in *R*. Also note we can compare two objects element-wise(`&` or `|`). Compare :

```
> sc = 1:10; sc1 = sc; sc1[10] = 0
> sc  == 1:10 && sc1 == 1:10
[1] TRUE
> sc  == 1:10 & sc1 == 1:10
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

The `&&` compares only the first element in `sc` with the first in `1:10`, and then the result with the comparison between `sc1[1]` and `1`, while the rest is not compared. On the other hand, the `&` compares every element and lets the user know exactly where the `FALSE` occured. The same goes for `||` and `|`.

2.11 Functions

We have actaully introduced you to a couple of built-in functions already. *Functions* have a name followed by a pair of parantheses where the user specifies *arguments* and *options*. *Arguments* are parameters that *must* be specified for the function to work, while *options* are simply optional. For example, if we wanted to find the mean of the group of numbers 1, 2, 3, 4 and 5, we might type:

```
> mean(c(1, 2, 3, 4, 5), na.rm = TRUE)
[1] 3
```

The *argument* for the function is the vector of numbers, represented by `c(1, 2, 3, 4, 5)`. We'll learn more about vectors later on. `na.rm = TRUE` is an *option* for the function, which tells the function to ignore any `NA` (Not Available) values. Many of the optional parameters have default values, which can be overwritten by the user. The arguments and options for each function are discussed in more detail in *R*'s built-in Help documentation.

R has most of the functions that you're going to need already built-in. However, *R* does allow you to write your own functions, which will be discussed later.

3 Plotting

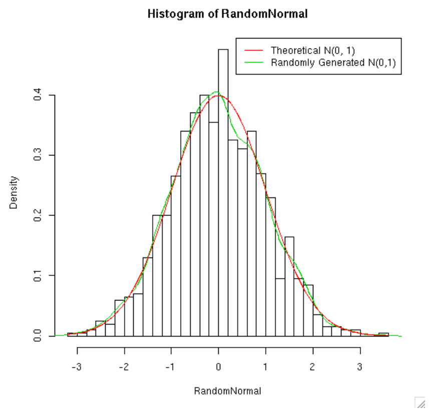
This section covers the basic graphical techniques to observing data.

3.1 Histograms

Suppose we wanted to see how well random generation of a normal variable with mean 0 and sd 1 does compared to the theoretical $Normal(0,1)$. We can see this by comparing plots of the theoretical distribution to that of our randomly generated data. The following code will yield the desired comparison, as seen in the graph below.

```
> RandomNormal = rnorm(1000, 0, 1)
> summary(RandomNormal)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-3.036000 -0.679200 -0.008942 -0.014720  0.669100  3.420000
> hist(RandomNormal, seq(-3.2, 3.6, .2), prob = TRUE)
> legend(legend = c("Theoretical N(0, 1)", "Randomly Generated N(0,1)" ),
+ col = c(2, 3), x = "topright", lty = 1)
> lines(density(RandomNormal, bw = .2), col = "green") # same as col = 3
> curve((1/sqrt(2*pi))*exp(-.5*(x)^2), add = TRUE, col = "red") # same as col = 2
```

Figure 2: Histogram :



The first line should be familiar. The second line is used to obtain upper and lower bounds for `seq` in the call to `hist` , in the third line. The `prob = TRUE` option displays the histogram in terms of the probability of each value in `seq(-3.2, 3.6, .2)`. In the call to the function `legend` we specify a vector of labels in `legend = c("Theoretical N(0, 1)", "Randomly Generated N(0,1)"` , colors with `col =c(2, 3)`, location for legend with `x=` and that we want lines in the legend by `lty =`. `lines` draws a line based on its arguments. `density` estimates the density of the data passed to it, `RandomNormal`, where `bw` is a smoothing bandwidth. It can be adjusted to make the line more or less smooth. `curve` is passed the density function of our theoretical $Normal(0,1)$ and the `add = TRUE` option to plot over the existing histogram.

3.1.1 colors

To see a list of colors recognized by R enter `colors()`, or `colours()`. Passing `col` a number in 1 through 8 will assign to `col` one of eight basic colors, and repeats for numbers greater than 8.

That is, *col = "black"* is the same as *col = 1 + 8k* for $k \in \mathbf{Z}_+$ Alternatively, passing `col` the value `colors()[i]` passes the i^{th} color name stored in `colors` to `col`. If the following list isn't enough variation for you, please see `colors()` for an extended list.

#	Color
1	`black`
2	`red`
3	`green`
4	`blue`
5	`aqua`
6	`pink`
7	`yellow`
8	`grey`

3.2 Box-Plots :

Box-plots can be used to compare the distributions of two variables, or data sets. Medians, 25% and 75% quantiles are shown on each graph for each variable along with a show of outliers in the data. For example, suppose we created the following data-frame

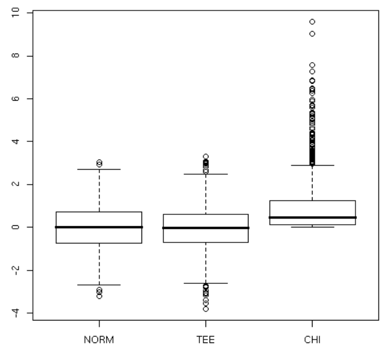
```
> DATA.oner = data.frame(NORM = rnorm(1000), TEE = rt(1000, 12), CHI = rchisq(1000,1))
```

Then we could create a box pot of the vectors in that data frame with `boxplot` as follows :

```
> boxplot(DATA.oner)
```

In the graph below the thick horizontal lines represent the respective medians, while the thin horizontal lines directly above and below are the 75% and 25% quantiles respectively. The lines above and below these represent a threshold beyond which points are considered outliers.

Figure 3: Boxplot of DATA.oner



3.3 Scatter-Plots :

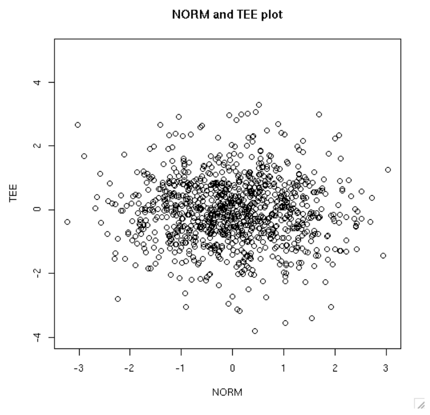
To plot data points from two vectors, we can use the `plot` function with the form

$$plot(vector_1, vector_2)$$

or similarly plot one vector by an index use the form $plot(vector_1)$. For example,

```
> plot(NORM, TEE, ylim = c(-4, 5), main = "NORM and TEE plot")
```

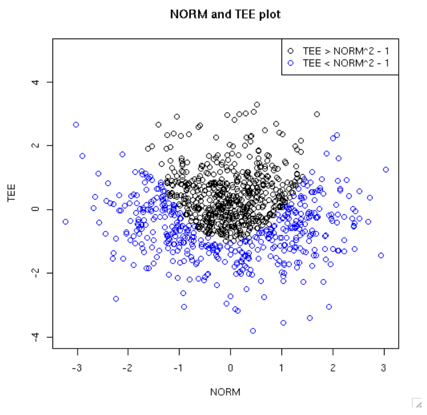
Figure 4: Scatter Plot :



The limits, or range, on which to plot can be specified in terms of `xlim = c(lowerBound, upperBound)` and `ylim = c(lowerBound, upperBound)` as can be seen in the call to `ylim` above. `main =` specifies the title of the plot. Labels may also be specified with the `xlab` and `ylab` options. We may superimpose points on top of our original plot with the `points` function. For example, to distinguish all points $(TEE_i, NORM_i)$ such that $TEE_i < NORM_i^2 - 1$ as blue points, we could call points as follws

```
> points(NORM[NORM^2 -1> TEE], TEE[ TEE + 1< NORM^2], col = "blue")
> legend(legend = c("TEE > NORM^2 - 1 ", "TEE < NORM^2 - 1" ),
+ col = c(1, 4), x = "topright", pch = 1)
```

Figure 5: Points on Scatter Plot



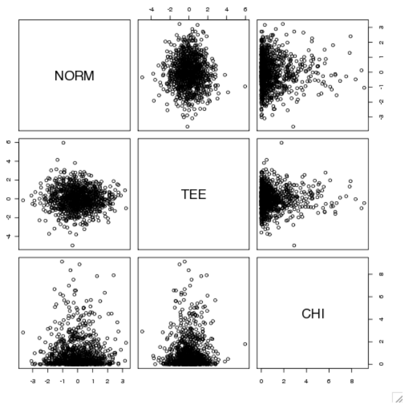
Note, the call to `pch = 1` tells *R* to use the point symbol in each label. You could also use `pch =c(1,1)`.

Lastly, if we wanted to plot all vectors of a data frame against each other, we could pass the name of the data frame to plot. For example,

```
> plot(DATA.oner)
```

yields the following plot

Figure 6: Plotting all vectors in a data frame :



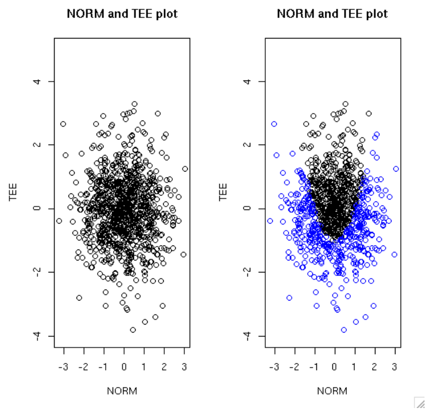
3.4 Putting Multiple Graphs in One Figure

3.4.1 par

Multiple graphs may be placed on one figure by using the `par` funcion along with the `mfrow` option. `mfrow` is used with the following format *mfrow = c(# rows, # columns)*. For example,

```
> par(mfrow = c(1, 2))
> plot(NORM, TEE, ylim = c(-4, 5), main = "NORM and TEE plot")
> plot(NORM, TEE, ylim = c(-4, 5), main = "NORM and TEE plot")
> points(NORM[NORM^2 -1> TEE], TEE[ TEE + 1< NORM^2], col = "blue")
```

Figure 7: A figure with two graphs :



3.4.2 split.screen

You should find that `split.screen` is more flexable than `par`. With `split.screen` you tell *R* how you want your figure divided, but then specifically state where you want each plot to go. Upon doing this, if you change your mind and only want to alter one graph, you can do this without starting the whole figure over from scratch. To begin the procedure, make sure that the `grid` library is loaded. You should also change the background to a non-transparant color. Even if your background looks white, do the following step before proceeding

```
> par(bg = "white") # set backgraound to non-transparant color
```

otherwise, you will end up plotting over existing plots when attempting to update screens.

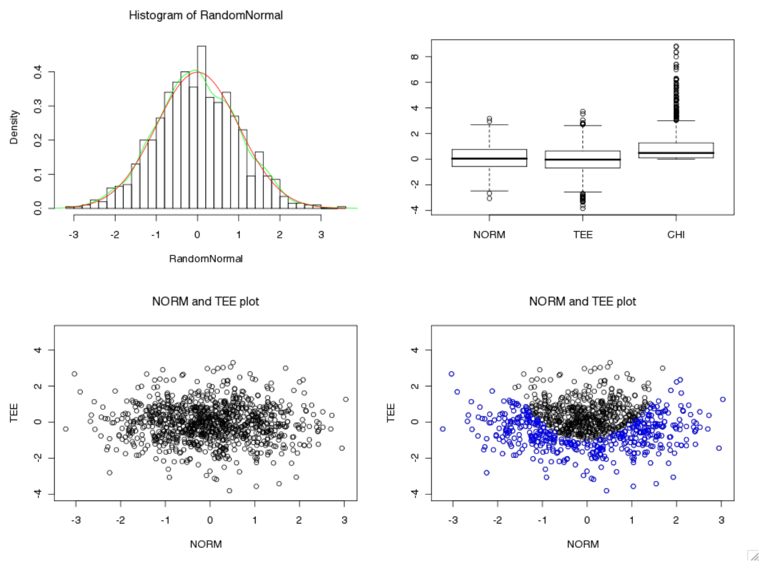
Then use `split.screen` with the following form *split.screen(c(# rows, # columns), screen to be split, erase = TRUE)*. We show its use through a couple of examples :

3.4.3 Replacing a Screen

Suppose we wanted all of the plots from this plotting section on one figure. We could do this with `split.screen` as follows

```
> screen.split(2,2)
[1] 1 2 3 4
screen(1) # what follows goes in slot (1,1)
> hist(RandomNormal, seq(-3.2, 3.6, .2), prob = TRUE)
> lines(density(RandomNormal, bw = .2), col = "green")
> curve(1/sqrt(2*pi)*exp(-.5*x^2), add = TRUE, col = "red")
>
> screen(2) # what follows goes in slot (1,2)
> boxplot(DATA.oner)
>
> screen(3) # similarly for (2,1)
> plot(NORM, TEE, ylim = c(-4, 5), main = "NORM and TEE plot")
>
> screen(4) # similarly for (2,2)
> plot(NORM, TEE, ylim = c(-4, 5), main = "NORM and TEE plot")
> points(NORM[NORM^2 - 1 > TEE], TEE[TEE + 1 < NORM^2], col = "blue")
```

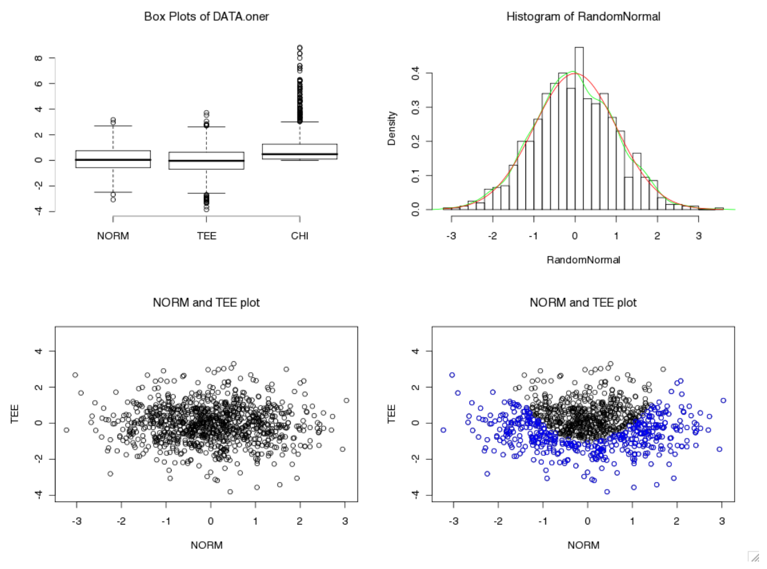
Figure 8: Multiple Plots via `split.screen`



Now, suppose you wanted the box plot in the upperleft slot and the histogram in the upper right and in addition you wanted to give a title to the boxplot. Simply reassign the contents of each screen accordingly

```
> screen(1) # Watch the screen (1,1) go blank
> boxplot(DATA.oner, main = "Box Plots of DATA.oner")
>
> screen(2) # again for slot (1,2)
> hist(RandomNormal, seq(-3.2, 3.6, .2), prob = TRUE)
> lines(density(RandomNormal, bw = .2), col = "green")
> curve(1/sqrt(2*pi)*exp(-.5*x^2), add = TRUE, col = "red")
```

Figure 9: Multiple Plots via `split.screen`

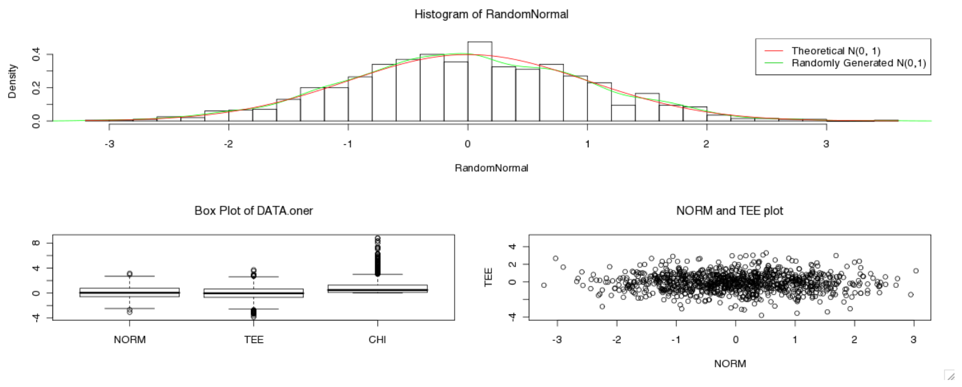


3.4.4 Split Screens within Split Screens

You may have noticed that the legend for the histogram was left out of the plot. That is becuase it is difficult to fit the legend in so small of a screen. Suppose, to solve this problem, we wanted a figure containing a large histogram and smaller box plot and scatter graph below. We can split screens that have already been split to achieve this goal. Observe the following example

```
> split.screen(c(2,1)) # The screen is now split in two
[1] 1 2
> split.screen(c(1,2), screen = 2) # splits the second into 2
[1] 3 4
> screen(1) # note screen 2 is now refered to as 3 and 4
> hist(RandomNormal, seq(-3.2, 3.6, .2), prob = TRUE)
> lines(density(RandomNormal, bw = .2), col = "green")
> curve(1/sqrt(2*pi)*exp(-.5*x^2), add = TRUE, col = "red")
> legend(legend = c("Theoretical N(0, 1)", "Randomly Generated N(0,1)" ),
+ col = c(2, 3), x = "topright", lty = 1)
>
> screen(3) # We call on screen 3 and 4, not 2
> boxplot(DATA.oner, main = "Box Plot of DATA.oner")
>
> screen(4)
> plot(NORM, TEE, ylim = c(-4, 5), main = "NORM and TEE plot")
```

Figure 10: Splitting Split Screens with `split.screen`

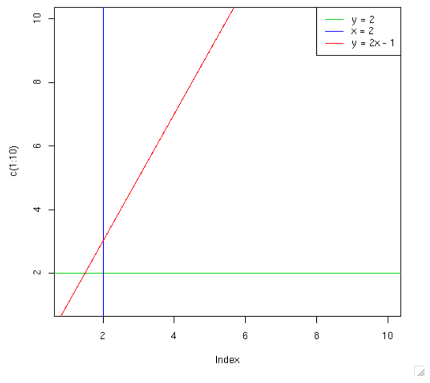


3.4.5 abline

To draw straight lines over an existing plot, use **abline** with the format *abline(a = intercept, b = slope, h = y value for horizontal line, v = x value for a vertical line)*. For example,

```
> plot(c(1:10), col = "white")
> abline(h = 2, col = "green")
> abline(v = 2, col = "blue")
> abline(-1,2, col = "red")
> legend(legend = c("y = 2","x = 2","y = 2x - 1"), col = c(3, 4, 2), lty = 1,
+ x = "topright")
```

Figure 11: lines $y = 2, x = 2$, and $y = 2x - 1$:

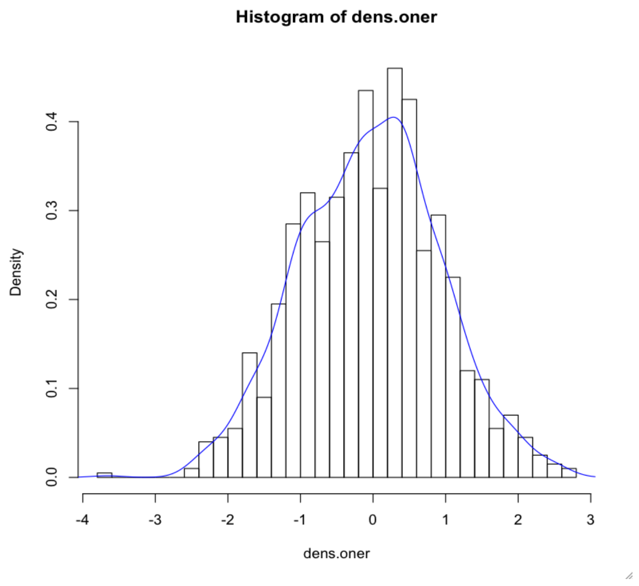


3.4.6 lines

To plot a line estimating the density of a vector over an existing histogram of that same vector, we use the `lines` function with the form `lines(density(vector), col = color of choice)`. For Example,

```
> dens.oner = rnorm(1000, 0, 1)
> max(dens.oner); min(dens.oner)
[1] 2.657019
[1] -3.660971
> hist(dens.oner, seq(-3.8, 2.8, .2), prob = TRUE)
> lines(density(dens.oner), col = "blue")
```

Figure 12: Using `lines` to plot the density of `data.oner`:



3.5 QQ-Plots :

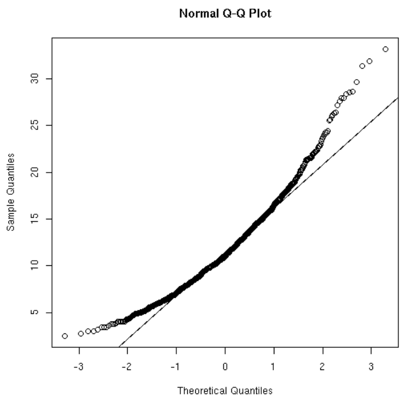
Quantile on Quantile plots, used to compare data-sets via comparison of quantiles, can be produced with the functions `qqplot` or `qqnorm` — the first comparing two vectors and the second comparing one vector to the appropriate normal distribution.

3.5.1 qqnorm :

To compare the quantiles of a vector to those of the $Normal(0, 1)$ distribution use `qqnorm` with the form `qqnorm(vector_1)`. To plot the line the graph should follow if the distributions are the same, you can use the `qqline` function, with the form $qqline(vector_1, datax)$. For example, to observe the relationships between a randomly created χ^2 with 12 degrees of freedom to the $Normal(0, 1)$, we could type

```
> CHI <-rchisq(1000, 12)
> qqnorm(CHI)
> qqline(CHI)
```

Figure 13: qqnorm plot :



3.5.2 qqmath (requires installation of `lattice` package) :

If you want to compare quantiles between your vector and a theoretical distribution that is not standard normal, you can use the `qqmath` function, found in the `lattice` package, with the form `qqmath(x = vector, distribution = q followed by distriution name)`. However, if your distribution requires a parameter, like degrees of freedom to the χ^2 distribution, then you must make your own function to pass to `qmath`. Don't worry, this is not difficult. For example, suppose we wanted to compare a randomly generated χ^2 with 12 degrees of freedom to the theoretical distribution. The first step is to define a function, say `qchisq.df12`, as follows

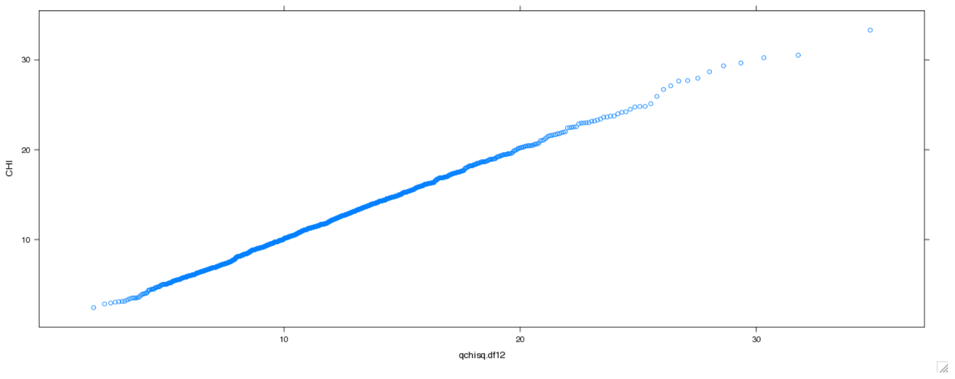
```
> qchisq.df12 = function(p) qchisq(p, df = 12)
```

Then we can pass `qchisq.df12` to `qqmath` as follows

```
> qqmath(x = CHI, distribution = qchisq.df12 )
```

Resulting in the following graph

Figure 14: qqmath plot :



3.5.3 qqplot :

To compare the quantiles of two vectors with `qqplot` the form is `qqplot(vector_1, vector_2)`. For example, to compare a vector named `D.set` with an unknown distribution to a random *exponential*($\lambda = .5$) we could use the `qqplot` correctly as follows

```
> EXP = rexp(1000, .5)
> qqplot(D.set, EXP)
```

3.5.4 Adjusting the qqline

If you try to plot a qqline on either of the non-`qqnorm` plots you will see that it doesn't work. The fix is really quite simple. If you ever feel the need to have a qqline on your qqplot, please look to the **Useful Functions** section for a discussion on how to create **QQ-lines** for these plots.

3.6 Linear Regression

Linear regression can be achieved using the function `lm` in *R*. The form is

```
lm(y ~ x_1 + x_2 + ... + x_n, data = name.of.data.frame)
```

where `y` is the dependent varaible and the `x`'s are independent variables. The `data` option is available in the case that you do not want to explicitly state `name.of.data.frame$x_i`, or `name.of.data.frame[, i]`, but instead want to use the above `x_i`. The function will give Ordinary Least Squares estimates for the coefficients corresponding the the `x`'s and an intercept term. We show how to use `lm` with an example.

Suppose we had the following data set, which is not real data, but randomly created with the `runif` function on 0 to 10 and 0 to 1.

```
> Dating.Data
         Men Intelligence Personality     Looks  Want.Date
1      Jimbo    1.5034497    9.172070 0.1458388 0.18518639
2  Billy-Bob    0.3673967    4.220902 5.1043472 0.57990543
3    Alfonso    9.5904054    5.865495 9.3900688 0.54772520
4         Li    3.1206422    1.941095 2.0819623 0.58571364
5        Bob    2.8993410    3.799475 8.8072740 0.01670105
6     Willie    2.6205701    8.098286 9.0606828 0.65386037
7      Vince    8.7863347    9.397780 4.2379685 0.96369422
8    Roberto    2.9002601    9.055264 9.7020046 0.77544638
9       Hugo    2.2993369    8.883192 2.3891803 0.27258566
10   Gerardo    6.8163340    4.719020 5.9723394 0.51717118
```

Assuming the second through fourth columns correspond to average rankings from a set of 100 women and the last column is the percentage of women willing to date each man, we could attempt to run a regression model for Intelligence, Personality, and Looks on the percent of women willing to date each man.

```
> lm(Want.Date ~ Intelligence + Personality + Looks, data = Dating.Data)->lm.Dating
```

and then get a summary of this regression with the `summary` function

```
> summary(lm.Dating)

Call:
```

```
lm(formula = Want.Date ~ Intelligence + Personality + Looks,
    data = Dating.Data)

Residuals:
     Min       1Q   Median       3Q      Max 
-0.42193 -0.19071  0.04056  0.23386  0.25035 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.17964    0.32309   0.556    0.598
Intelligence  0.03360    0.03343   1.005    0.354
Personality   0.02176    0.03718   0.585    0.580
Looks         0.00896    0.03040   0.295    0.778

Residual standard error: 0.3032 on 6 degrees of freedom
Multiple R-squared: 0.2278,     Adjusted R-squared: -0.1583 
F-statistic: 0.5901 on 3 and 6 DF,  p-value: 0.6437
```

and similarly, get an analysis of variance table with `anova`

```
> anova(lm.Dating)
Analysis of Variance Table

Response: Want.Date
             Df  Sum Sq Mean Sq F value Pr(>F)
Intelligence  1 0.12546 0.12546  1.3647 0.2870
Personality   1 0.02930 0.02930  0.3187 0.5928
Looks         1 0.00798 0.00798  0.0869 0.7781
Residuals     6 0.55156 0.09193
```

Now, we can observe if the OLS assumptions hold. Recall, there should be a constant variance for the residuals and no correlation between these and the independent variables. We can observe correlation with the `cor` function

```
> cor(Dating.Data[,c(-1, -5)], lm.Dating$residuals)
                      [,1]
Intelligence -8.661888e-17
Personality   1.473017e-16
Looks         5.867616e-17
```

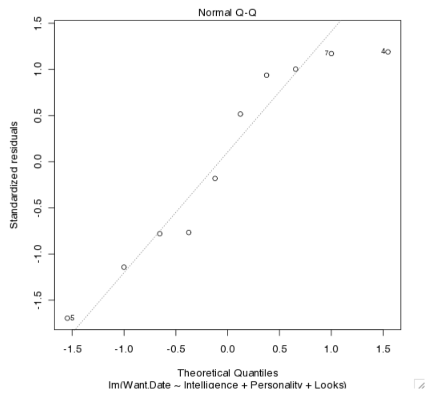
These are essentially zero, so one assumption holds. How about correlation between independent varaibles?

```
> cor(Dating.Data[,c(-1, -5)])
             Intelligence Personality       Looks
Intelligence   1.00000000  0.06213098  0.27256812
Personality    0.06213098  1.00000000 -0.05558863
Looks          0.27256812 -0.05558863  1.00000000
```

We can observe the assumption of independent and identically distributed (iid) residuals, by plotting `lm.Dating` and hitting the `Return` key until we obtain the `qqnorm` plot of standardized residuals.

```
> plot(lm.Dating)
Hit <Return> to see next plot:
Hit <Return> to see next plot:
Hit <Return> to see next plot:
```

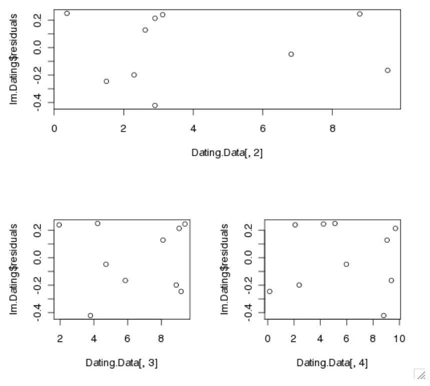
Figure 15: qqnorm Plot of Standardized Residuals :



This hovers around the QQ-line, but also has outliers. So an assumption is violated. Lastly, to test for constant variance, you could plot the residuals against the independent variables.

```
> split.screen(c(2,1))
[1] 1 2
> split.screen(c(1,2), screen = 2)
[1] 3 4
> screen(1)
> plot(Dating.Data[,2], lm.Dating$residuals)
> screen(3)
> plot(Dating.Data[,3], lm.Dating$residuals)
> screen(4)
> plot(Dating.Data[,4], lm.Dating$residuals)
```

Figure 16: Plots of Residuals Against Independent Variables:



3.7 Saving Plots

To save plots into pdf files, you can use a combination of the `pdf` function and the `dev.off` function. The format for `pdf` is `pdf(file = "file name to save plot to")`. There are many other options to the `pdf` function we will not be covering, so we suggest observing the help page for this function if you want more flexible use of the function. We then enter the commands for the plots we want to have in the pdf. When finished with the commands, we enter `dev.off()` to close the process and finish creation of the file. To save, for example, the plots from a linear regression model, say `lm.Dating`, we would do the following

```
> pdf(file = "Dating.Regression.Plots.pdf")
> plot(lm.Dating)
> dev.off()
null device
          1
```

Then looking for this file in the current directory, we would find all plots from the regression plot, one per page.

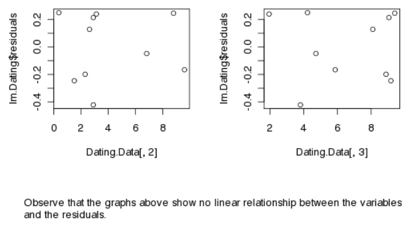
Please note that there are similar functions for `png`, `jpeg`, `bmp`, `tiff`, and other formats. As usual, use one of the help functions to see if a function relating to another format exists.

3.7.1 Adding Text :

We can actually write a report in pdf format using `split.screen`, `pdf` and `mtext` to plot and write text directly into pdf files, but this is not very practical. However, for small comments to plots `mtext` can be very useful. The format is `mtext("place text here, or a character vector", line = 0, adj = NULL)`. Please see the help page for this function for a detailed description. We show its use in an example.

```
> split.screen(c(2,1))
[1] 1 2
> split.screen(c(1,2), screen = 1)
[1] 3 4
> screen(1)
> screen(3)
> plot(Dating.Data[,2], lm.Dating$residuals)
> screen(4)
> plot(Dating.Data[,3], lm.Dating$residuals)
> screen(2)
> mtext("      Observe that the graphs above show no linear relationship
+ between the variables \n      and the residuals.", line = 0, adj = 0)
```

Figure 17: An example of using `mtext`:



Observe that the graphs above show no linear relationship between the variables
and the residuals.

Here, we make use of the newline character, $\backslash n$. The placing of text can be altered with `line`, increasing the height of the starting line by using `line` > 0, and decreasing the height with `line` less than zero. Leaving `adj` in its default setting centers the text, while setting it to `0` aligns the text to the left.

4 Packages

4.1 Introduction

In *R*, built-in functions are stored in packages — collections of functions, and sometimes datasets. Some come with *R* upon installation — standard packages — while others you must obtain manually. We discuss how to install and load packages for use in *R*.

4.2 Listing Loaded and Installed Packages

To observe the standard packages enter the following after beginning your session

```
>search()
[1] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "methods"
[7] "base"
```

These packages are always available upon starting an *R* session. We will add to this list in the following sample sessions. However, each time a new *R* session is started the libraries loaded will be reset to the above list. (We introduce a function to make reloading packages a little easier in the **Function** section.)

4.2.1 .packages

To list all *R* packages you have installed, use the `.packages` function along with the option `all.available = TRUE`. (If you have not installed any packages yet, feel free to skip ahead to the Listing All *R* Packages section, install a couple and come back to this section. Note how the list from `.packages` is different from `search` before loading of new packages.)

```
> .packages(all.available =TRUE)
 [1] "CGIwithR"       "DBI"            "KernSmooth"     "MASS"
 [5] "RMySQL"         "RUnit"          "Rmetrics"       "XML"
 [9] "adapt"          "base"           "boot"           "class"
[13] "cluster"        "codetools"      "datasets"       "fArma"
[17] "fAsianOptions"  "fAssets"        "fBasics"        "fCalendar"
[21] "fCopulae"       "fEcofin"        "fExoticOptions" "fExtremes"
[25] "fGarch"         "fImport"        "fMultivar"      "fNonlinear"
[29] "fOptions"       "fPortfolio"     "fRegression"    "fSeries"
[33] "fTrading"       "fUnitRoots"     "fUtilities"     "foreign"
[37] "grDevices"      "graphics"       "grid"           "lattice"
[41] "lpSolve"        "maps"           "methods"        "mgcv"
[45] "mnormt"         "nlme"           "nnet"           "polspline"
[49] "quadprog"       "rcompgen"       "robustbase"     "rpart"
[53] "sn"             "spatial"        "splines"        "stats"
[57] "stats4"         "survival"       "tcltk"          "tools"
[61] "urca"           "utils"          "zoo"
```

4.2.2 library

Alternatively, a page containing descriptions of each package installed can be obtained with the `library` function as follows

```
>library()
```

If you are not operating out of the *R* Graphical User Interface (GUI), but instead from your computer's command line on a Unix system, then to search for a specific word on the page, enter `/` followed by your keyword. For example, `/base` will highlight all instances of `base` on the page. Enter `q` to exit the page.

Use the `library` function along with the `help =` option to look at all functions in a particular package. To do this for `"survival"`, type

```
>library(help = "survival")
```

A page titled `Information on the package "survival"` will appear, listing every function in the package. At the top of the page is a description of the package. Note the line

```
Depends:       stats, utils, graphics, splines, R (>= 2.0.0)
```

Compare these packages to the ones listed earlier in calling `search()`.

In order to make a package available for use, use the `library` function with the general form *library(package of interest)*. Upon doing this, the package of interest will show up in `search()` for the rest of the session. For example, to make `survival` available for use, type

```
> library(survival)
Loading required package: splines
```

Note that `survival` and `splines` are now listed with `search()`

```
> search()
 [1] ".GlobalEnv"        "package:survival"  "package:splines"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "package:methods"
[10] "Autoloads"         "package:base"
```

4.3 Listing all *R* packages

To get a complete listing of all *R* packages you can use the `available.packages()` function

```
> available.packages()
--- Please select a CRAN mirror for use in this session ---
Loading Tcl/Tk interface ... done
```

4.4 Installing Packages

4.4.1 From the Command Line :

Upon entering `available.packages()` you will be prompted to choose a Comprehensive R Archive Network (CRAN) mirror. This is basically a website or collection of files containing R documentation copied from the original CRAN server so that we can access the documentation quickly from the closest mirror. Naturally, you should choose the location closest to your own. Then the list will be displayed. To install a package, use the `install.packages` function. For example, to install the `xtable` package, type

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

Don't forget to use `library(xtable)` to make the package available for use in your *R* session.

4.4.2 Using a Graphical User Interface (GUI)

If you are not using the command line, in one way or another, to navigate through your *R* session, you are using the *R* application GUI. In this case, you can use a "point and click" approach to installing packages. Find the `Packages and Data` or `Packages` option and choose `Package Installer`. Another GUI should appear called `R Package Installer`. To find a package, enter a keyword in the search path, click `Get List` to display a list matching your keyword. Find the package of interest and then click the `Install Selected` button. If you look at your *R* GUI you should see a lot of script rolling by the page. When this finishes, the last line should display the location where the package(s) have been stored.

If you are not the administrator on the system you are operating out of, use the `At User Level` option.

4.5 Removing Packages

To remove a package, use the `remove.packages` function with the form *remove.packages(package to remove, directory where package is stored)*. To remove the `xtable` package, for example, type

```
> remove.packages("xtable", "/Library/Frameworks/R.framework/Resources
+ /library/xtable")
```

Leaving out the second argument yields the following warning :

```
> remove.packages("xtable")
Warning in remove.packages("xtable") :
  argument 'lib' is missing: using /Library/Frameworks/R.framework/Resources/library
```

Unless you specify otherwise, the location of the package should be in
`/Library/Frameworks/R.framework/Resources/library/put-package-name-here`

5 Help R help you

In *R* there are many sources of help to the user. The importance of knowing how to access information about *R* through its help pages cannot be stressed enough. Without a firm grasp of them, you will no doubt be lost. We cover the main help options and when to use them.

5.1 help.start

The `help.start()` help option gives an html version of *R* documentation. Not only is this more interactive than `help` with links to manuals on similar inquiries not available through command line help, it also offers tutorials, lists of packages and other information at the click of a link. To start the `help.start()` mode, simply type

```
> help.start()
```

5.2 help

In *R* different functions are stored in libraries, or packages. To see which ones are available in your session enter `search()`. If you want information on a command (or summary of commands) you know exists in any of these libraries, use the following format :

?command

OR

help("command")

For example, to get help on the mean function, type

```
>?mean
```

OR

```
>help(mean)
```

A page will appear telling you all about the mean function. As with `library()`, if operating out of the terminal, press `q` to exit the page. Similarly, in this case, to search for a word or phrase in the help page type

```
/word or phrase
```

followed by the `Enter` or `Return` key. Recall, the `/` puts the page into a search mode, searching for what follows the slash on the page.

However, if your query is not in any of the packages or libraries currently available for use in your session (but has been installed) then help will not be able to locate the relevant information unless you use the option `try.all.packages = TRUE`. For example :

```
> ?RollingAnalysis
No documentation for 'RollingAnalysis' in specified packages and libraries:
you could try 'help.search("RollingAnalysis")'
```

Before calling on `help.search` we can try using the `try.all.packages= TRUE` option whereby the help function tells us where to find the package necessary for use of `RollingAnalysis`, given this is the exact name of some help page.

```
> help(RollingAnalysis, try.all.packages = TRUE)
Help for topic 'RollingAnalysis' is not in any loaded package but can
be found in the following packages:

  Package               Library
  fTrading              /Library/Frameworks/R.framework/Resources/library
```

To make the package available for use, type the below line. After doing so, `help("Rolling Analysis")` will work.

```
  > library(fTrading)
```

In general, the form is *library(package of interest)* where the package of interest has already been installed.

5.2.1 Graphical User Interface (GUI) Specific Tip

5.2.2 Mac OS X

If working out of the *R* GUI, using `help(query)` will open a page for `query` if this is the name of a built in loaded function, or the name of a help page corresponding to a loaded package. Similarly, if it is not, then *R* will tell you this and the search will be over. However, if you open the help page for a function you know exists, like, say, `help(mean)`, then type your inquery in the search engine on the GUI help page, you will notice that it matches keywords just as `help.search` does. That is, by always keeping the GUI help page open, you always have access to `help.search` without having to type help.search each time you want to use it.

5.2.3 Windows OS

On a Windows operating system the idea is similar, but searching is not akin to `help.search` as much as it is to a list of topics similar to the query. Nonetheless, you may find it worthwhile to just keep the help page open throughout your session.

5.3 apropos and find

Alternatively, if you know a keyword that is a part of the function's name you can use the `find` and `apropos` functions in combination to search for functions from installed packages containing that keyword. The general format is :

apropos("keyword", what = TRUE/FALSE, mode = "mode of the keyword")

find("keyword", simple.words =TRUE/FALSE, numeric = TRUE/FALSE)

Using the `what = TRUE` will list the accompanying packages' locations in `search()`, while allowing the default `what = FALSE` omits this. The `mode` option allows you to specify the mode of the object you are looking for. By default, this is set to `"any"`.

5.3.1 searching for functions and their packages

If you are interested in knowing a summary where each function resides, a general strategy is to use `find` to list the packages containing the keyword, then use `apropos` to list the matching functions. We can easily use the `where = TRUE` to match. Lets look for packages containing the keyword "mean" :

```
> find("mean", simple.words = FALSE, numeric = TRUE)
        .GlobalEnv      package:rpart package:fUtilities        package:zoo
                 1                  2                  9                 10
     package:stats       package:base
                17                 24
> apropos("mean", where = TRUE)
                 1                  9                 24                 17
   "boot.mean.rep"         "colMeans"         "colMeans"           "kmeans"
                24                 24                 24                 24
            "mean"        "mean.Date"     "mean.POSIXct"     "mean.POSIXlt"
                24                 24                 24                  9
 "mean.data.frame"     "mean.default"    "mean.difftime"  "mean.timeSeries"
                 2                  3                 10                 10
         "meanvar"         "rollMean"         "rollmean" "rollmean.default"
                 9                 24                 17
        "rowMeans"         "rowMeans"    "weighted.mean"
```

Now, if we were interested in `weighted.mean`, we would match `17`, so it is in the `stats` package. If we just wanted the exact name of the function, we could have typed

```
> apropos("mean")
```

then used the help function.

5.3.2 searching for lost objects :

From time to time you will misplace an object you have created. However, typically, you will be able to recall the mode of this object — ie, was it a list, a vector of characters, numbers, etc? For example, suppose you had recently created an object and all you can recall about it is that it's a vector of mode numeric and has the string "icker", in its name. Could be tickers, bickers, ickers, etc. By calling `apropos("icker", mode = "numeric")` you can get a reduced list for you query if there happens to be other non-numeric mode objects with "icker" in the name. For example, the following code lists 27 matches to `apropos("ickers")` but only one to `apropos("ickers", mode = "numeric")`

```
> apropos("icker"); apropos("icker", mode = "numeric")
 [1] "TICKERS"   "ickers.1"  "tickers.A" "tickers.B" "tickers.C" "tickers.D"
 [7] "tickers.E" "tickers.F" "tickers.G" "tickers.H" "tickers.I" "tickers.J"
[13] "tickers.K" "tickers.L" "tickers.M" "tickers.N" "tickers.O" "tickers.P"
[19] "tickers.Q" "tickers.R" "tickers.S" "tickers.T" "tickers.U" "tickers.V"
[25] "tickers.W" "tickers.X" "tickers.Y"
[1] "ickers.1"
```

5.4 help.search

The limitations of `apropos` and `find` are that (1) the object name must contain the keyword and (2) it only searches in the loaded packages. Recall, to see what packages are installed type

.packages(all.available = TRUE or library(). With help.search all *R* installed packages will be searched. If you have no idea about the name of a command but know a keyword relating to it use the help.search function which has the general format :

help.search("command")

Upon doing this, a screen will appear listing all matches to your inquery, *"command"* in this case. The list will consist of functions or the name of the help page where the function can be found. Each is accompanied by a description of its use along with the name of the library necessary for its use in parentheses. This can be seen in the example below. Further note that when help.search("mean") is typed, functions such as colSums appear that did not with the use of apropos("mean").

```
>help.search("mean")
Help files with alias or concept or title matching
'mean' using regular expression matching:



DateTimeClasses(base)   Date-Time Classes
Date(base)                    Date Class
colSums(base)            Form Row and Column Sums and Means
difftime(base)              Time Intervals
mean(base)                 Arithmetic Mean
sunspot(boot)              Annual Mean Sunspot Numbers
meanabsdev(cluster)      Internal cluster functions
tmd(lattice)             Tukey Mean-Difference Plot
meanvar(rpart)           Mean-Variance Plot for an Rpart Object
kmeans(stats)            K-Means Clustering
oneway.test(stats)       Test for Equal Means in a One-Way Layout
weighted.mean(stats)     Weighted Arithmetic Mean



Type 'help(FOO, package = PKG)' to inspect
entry 'FOO(PKG) TITLE'.
```

Scrolling down the help page shows many more functions matching the query. We see that the mean function requires the base library, which is already available upon starting R. However, observe meanpart, the 9th function on the list. This requires the library rpart. Let's try to call meanvar :

```
>meanvar
Error: object "meanvar" not found
```

Now, lets try it again after making the rpart library available for use :

```
>library(rpart)
>meanvar
function (tree, ...)
UseMethod("meanvar")
<environment: namespace:rpart>
```

Which in simplest terms means *R* now recognizes the function `meanvar` (and all others now in the `rpart` library). Now we can use the help command to take a more in depth look at the function `meanvar`. Type the following and see what you get :

```
>help(meanvar)
```

If you want to narrow your search to a specific package, use the `package =` option. For example, if you were confident that the function giving means and sums of the columns and rows of a data-frame was in the base library, you could write the below code and get the following limited list:

```
> help.search("mean", package = "base")

DateTimeClasses(base)   Date-Time Classes
Date(base)              Date Class
colSums(base)           Form Row and Column Sums and Means
difftime(base)          Time Intervals
mean(base)              Arithmetic Mean
```

5.5 RSiteSearch

Lastly, there is the option of searching uninstalled packages online through the `RSiteSearch` function. To search use the form `RSiteSearch("query")`. There are many options available for this function that can be stated explicitly in the call to `RRSiteSearch` or accessed trough the browser it calls. In particular, at this level, there are pleanty of interesting pages on datasets available upon installation of neccessary packages.

5.6 Summary

In summary, if you know the exact name of the function and think it is in a loaded package, use `help`. If not in a loaded package, try the option `try.all.packages = TRUE`. If you don't know the exact name of the function, but know it is in an installed package, use `apropos` — along with `find` if you need to know the accompanying package. If you are not sure of the name but have a keyword on the gereral operations of the function, use `help.search` — along with `package =` if you have an idea of what package the function resides in. Using this strategy, you should be able to answer most questions you have about functions on your own.

6 Reading Data Into *R*

Most data can be read into *R* through the use of one of the read functions or through `scan`. We cover `read.table`, `scan`, an exmple of analyzing data, and an example of cleansing data in *R* in this section. While cleaning may often be done outside of *R* in a text editor, it is good to know how to do it in *R*. It is also worth noting that the `scan` function reads the data in as a vector. In turn, we show an example of reorganizing this data into an appropriate data-frame form, when the data is not intended to be a single vector.

6.1 read.table

R has a workhorse in the function `read.table`. In short, it reads data into *R* from a file or url according to your specifications. The general format at this level is (note the quotes around the filename or url):

```
read.table("filename or url", header = FALSE, sep = "", skip = 0 , dec = ".", row.names,
col.names, nrow, stringsAsFactorso = TRUE)
```

While a filename or url must be specified, the rest is optional, available to meet your particular needs. Here `sep` tells `read.table` how to separate the entries. `dec = "."` indicates that in the data the decimal is ".". The `skip` option describes how many lines to skip. `col.names` (`row.names`) is a vector containing names for the columns (rows) of the data-frame. `header` will tell `read.table` if the column names are already given in the data. If so, just set `header = TRUE`. `nrow` tells the function when to stop reading rows of data. Lastly, `stringsAsFactors` set to `TRUE`, by default, will change any strings in the data to factors, while setting it to `FALSE` will read in strings as `character` types.

In the rest of this section, we first show a simple example of inputting data followed my an example of possible statistical analysis of this data. Secondly, we show a more complex example of reading data with `read.table` and `scan`.

Assume we had the following file, *poundData.txt*, containing the days that dogs of different breeds resided at particular pounds :

```
dogs,pound.A,pound.B,pound.C
Pit-Bull,124,64,46
Jack-Russell-Terrier,35,13,43
Akita,102,81,100
German-Shephard,51,19,56
Pug,23,30,17
Afghan-Hound,129,48,64
Beagle,54,6,13
Basset-Hound,123,17,92
Cocker-Spaniel,144,80,98
Austrailian-Shephard,21,7,47
```

We can read this data into *R* many ways. One way to do this and then display the contents is to do the following :

```
> pet.data = read.table("poundData.txt", sep = ",", header = TRUE)
> pet.data
                  dogs pound.A pound.B pound.C
1             Pit-Bull     124      64      46
```

```
2  Jack-Russell-Terrier      35       13       43
3                 Akita     102       81      100
4       German-Shephard      51       19       56
5                   Pug      23       30       17
6          Afghan-Hound     129       48       64
7                Beagle      54        6       13
8          Basset-Hound     123       17       92
9        Cocker-Spaniel     144       80       98
10 Austrailian-Shephard      21        7       47
```

You can check to see that `pet.data$dogs` is indeed a `"factor"`, by entering `class(pet.data$dogs)`. We can change this with

```
> pet.data$dogs = as.character(pet.data$dogs)
```

or by originally using `stringsAsFactors = FALSE` when reading in the data with `read.table`.

We could then proceed to do statistical analyses on the data. A nice first step is to use the `summary` function on the data

```
> summary(pet.data)
                 dogs       pound.A          pound.B          pound.C      
 Afghan-Hound        :1   Min.   : 21.0   Min.   : 6.0   Min.   : 13.00  
 Akita               :1   1st Qu.: 39.0   1st Qu.:14.0   1st Qu.: 43.75  
 Austrailian-Shephard:1   Median : 78.0   Median :24.5   Median : 51.50  
 Basset-Hound        :1   Mean   : 80.6   Mean   :36.5   Mean   : 57.60  
 Beagle              :1   3rd Qu.:123.8   3rd Qu.:60.0   3rd Qu.: 85.00  
 Cocker-Spaniel      :1   Max.   :144.0   Max.   :81.0   Max.   :100.00  
 (Other)             :4
```

Assume for a moment that the dogs' types were unknown and that the dogs were just picked at random from the pounds — that is, assume we didn't have the first column of data. Then considering we don't know the distribution of the dogs' time in each pound, we may be interested in ranking the observances, using the `rank` function, and then finding the means of ranks as follows

```
> Rankings = rank(c(pet.data[,2], pet.data[,3], pet.data[,4]))
> # use Rankings = rank(sapply(sapply(pet.data[,-1], as.matrix), as.matrix))
> # if you have many columns, or treatments
> Rankings.A = Rankings[1:length(pet.data[,1])]
> Rankings.B = Rankings[(length(pet.data[,1]) + 1) : (2*length(pet.data[,1]))]
> Rankings.C = Rankings[(2*length(pet.data[,1]) + 1) : (3*length(pet.data[,1]))]
> mean(Rankings)
[1] 15.5
> mean(Rankings.A)
[1] 20.1
> mean(Rankings.B)
[1] 10.65
> mean(Rankings.C)
[1] 15.75
```

In the code above, recall that `c` combines its arguments to form one vector. Since `Rankings` has elements one through 10 from `pound.A`, elements 11 through 20 from `pound.B`, and elements 21 through 30 from `pound.C`, we can just pull the appropriate elements from `Rankings` to get the overall rankings of the elements from each of the three pounds. In turn, `Rankings.A` is defined by simply indexing the first 10 elements (`1:length(pet.data[,1])`) of `Rankings` and assigning these values to `Rankings.A`; and a similar pattern holds for the others — `Rankings.B` and `Rankings.C`. Please observe what follows, even if it is above your knowlege of statistics for appreciation of the built-in function `kruskall.test`

We could then proceed to do a non-parametric test on the data, such as a Kruskal Wallis Test, to see if there is a significant difference in the means of the times at each pound, in one of two general ways. The first is to calculate the approximately $\chi^2_{\#Treatments-1}$ statistic $K = \frac{12SS_B}{N(N+1)}$, where N is the total number of observances, 30. SS_B is the sum of squared differences between each treatement's mean rank and the overall mean rank, times the number of observances per treatment.

```
> mean(Rankings.A) -> M.R.A
> mean(Rankings.B) -> M.R.B
> mean(Rankings.C) -> M.R.C
> mean(Rankings) -> M.R.Overall
> K = 12*(10*(M.R.A - M.R.Overall)^2 + 10*(M.R.B - M.R.Overall)^2 +
+ 10*(M.R.C - M.R.Overall)^2)/(30*31)
> K
[1] 5.773548
```

We could then see if the statistic falls in the acceptance region with a Type I error level (or significance level) of 0.05 : $(\chi^2_2(0.025), \chi^2_2(0.975))$ as follows

```
> K < qchisq(.975, 2) & K > qchisq(.025, 2) # see the section on Probability
[1] TRUE
```

This literally asks the question "Is K between the .025 and .975 quantiles of the χ^2 distribution with two degrees of freedom, TRUE or FALSE?". The answer is `TRUE`, so K is in the acceptance region. So, we would not reject the null hypothesis at the 5% significance level that the distribution of times in the different pounds have the same location parameters. Here `&` stands for the logical operator "AND". Note that `|` stands for "OR". So, to ask "Is K not between the 0.025 and 0.975 quantiles of the χ^2 distribution with 2 degrees of freedom" we could ask

```
> K >= qchisq(.975, 2) | K <= qchisq(.025, 2)
[1] FALSE
```

So, the test statistic is not in the rejection region, as we would hope given the previous answer. We could also get a P-value, using the cumulative density function pchisq to see the probability of seeing the `K` value we did or something more extreme, as follows

```
> pchisq(K, 2)
[1] 0.9442442 # is greater than 0.5, so we want 1 - pchisq(K, 2)
> 1 - pchisq(K, 2)
[1] 0.05575578 # our P-value
```

The second approach would be to use the built-in Kruskal Wallis test. Try finding it with `help.search`. After you do, (or, don't) observe that the work above is done with one line of code using the built-in function.

```
> kruskal.test(pet.data[,-1])

        Kruskal-Wallis rank sum test

data:  pet.data[, -1]
Kruskal-Wallis chi-squared = 5.7774, df = 2, p-value = 0.05565
```

You may alternatively want to observe the data using categorical tests. Try this as an exercise if you have the background.

6.1.1 setwd

You can change your working directory in your R session by using the `setwd` command. For example, suppose we were on a Unix system in our home directory `/`, but *poundData.txt* was actually located at

/Users/JoeShmoe/Animal.Data/poundData.txt.

By originally typing

```
setwd("/Users/JoeShmoe/Animal.Data")
```

we could then proceed as above by passing `poundData.txt` to read.table, as opposed to passing the entire location to `read.table`. This is very useful if you keep many data sets in one directory.

6.1.2 An Example of Cleaning Data

Assume we had a small file *Jimbos.baseball.stats.txt* consisting of the following

```
Jimbo’s Baseball Statistics
Year    BA       AB      BB      H       R       RBI  SO  HR
1988    .300      10       2       1       3       1   3   1
1989    .300     300      60      90      30      60  90  30
1990*   .313     400      80     125     100*    120* 100 60*
1991    .325     400      80     130      70     100  120 40
*There is much controvercy surrounding Jimbo’s 1990 season.
```

We see that the first line of the file contains `Jimbo’s Baseball Statistics`, which we do not want to read into R, so we will tell `read.table` to skip one line in reading the data with `skip = 1`. Further, we see that the columns already have names. Why not just use these for the column names of our dataset? We can do this with the `header = TRUE` option. Lastly, we will get an error message when reading data into R if we don’t deal with the last line `*There is much ...`. To deal with this we tell R to stop reading the data after four rows have been read — recall the first line was ignored and the second used as a header, so the count starts with the first observance, or row of data in our case. We tell `read.table` to stop reading after four rows with `nrow = 4` option. We can now use these options to read the data into R with the following code :

```
>Jimbo.data.1 = read.table("Jimbos.baseball.stats.txt", skip = 1, header = TRUE, nrow
= 4)
```

Checking the data-frame Jimbo.data.1, we have :

```
> Jimbo.data.1
   Year    BA  AB BB   H    R  RBI  SO  HR
1  1988 0.300  10  2   1    3    1   3   1
2  1989 0.300 300 60  90   30   60  90  30
3 1990* 0.313 400 80 125 100* 120* 100 60*
```

```
4  1991 0.325 400 80 130   70  100 120  40
```

Now we run across the problem of making computations involving the third row or 6^{th}, 7^{th}, and 9^{th} columns, which will be recognized as `factors` by R. As a solution, we could do the following:

```
> Jimbo.data.1 = read.table("Jimbos.baseball.stats", nrow = 4, skip
+ = 1, header = TRUE, stringsAsFactors = FALSE)
> Jimbo.data.1[3,] = gsub("[^.0-9]","",Jimbo.data.1[3,])
> Jimbo.data.1 = sapply(Jimbo.data.1, as.numeric)
> Jimbo.data.1 = data.frame(Jimbo.data.1)
> Jimbo.data.1

     Year    BA  AB BB   H    R  RBI  SO  HR
1  1988 0.300  10  2   1    3    1   3   1
2  1989 0.300 300 60  90   30   60  90  30
3  1990 0.313 400 80 125  100  120 100  60
4  1991 0.325 400 80 130   70  100 120  40
```

The Details :

We read the data, as is, into R and then use built in functions to remove the "*"'s from row 3. This will involve manipulation of characters, or strings of characters, so we can make things easier on ourselves by reading the * data into R as characters, with the `stringsAsFactors = FALSE` option for `read.table`.

```
> Jimbo.data.1 = read.table("Jimbos.baseball.stats", nrow = 4, skip = 1, header = TRUE,
stringsAsFactors = FALSE)
```

Next, we use the built in function `gsub` which has the following form :

```
gsub("text to be replaced", "text to substitute its place", Vector of interest)
```

Along with a regular expression $[.\wedge 0-9]$. Placing $\wedge$ inside of brackets tells R we do not want what is to follow the $\wedge$. In this case, $[\wedge.0-9]$. $0-9$ represents the numbers zero through nine. Hence in passing this regular expression to the `pattern to be replaced` section of `gsub`, we are replacing anything in the vector of interest, `Row 3`, that is not a number or a dot. This will rid the data of the *'s.

```
> Jimbo.data.1[3,] = gsub("[^.0-9]","",Jimbo.data.1[3,])
```

Lastly, we can use the `sapply` function to convert the character type columns of our data set into numeric columns. The basic form of `sapply` is :

```
sapply(Data to perform a function functionName on, functionName)
```

```
> Jimbo.data.1 = sapply(Jimbo.data.1, as.numeric)
```

Since `sapply` turns Jimbo.data.1 into a matrix, this last step turns Jimbo.data.1 back into a dataframe.

```
> Jimbo.data.1 = data.frame(Jimbo.data.1)
> Jimbo.data.1


    Year    BA  AB BB   H    R  RBI  SO  HR
1  1988 0.300  10  2   1    3    1   3   1
2  1989 0.300 300 60  90   30   60  90  30
3  1990 0.313 400 80 125  100  120 100  60
4  1991 0.325 400 80 130   70  100 120  40
```

6.2 The Rest of the read Family:

The following table summarizes the read functions that will probabaly be of interest to you. Note "\\t" is tab. Each has a varaition in the `sep =` and `dec =` options from `read.table`

`read.`	`sep =`	`dec =`
`csv`	`","`	`"."`
`csv2`	`";"`	`","`
`delim`	`"\t"`	`"."`
`delim2`	`"\t"`	`","`

For example, if your data is separated by tabs, but the decimals are commas, and you don't want to explicitly state this in `read.table`, you could just use `read.delim2`. There are many more read functions in *R*. Another read function of interest is `ReadLines`. To find out more about this and other read functions, use `apropos("read")` and then the `help` function accordingly.

6.3 scan

Alternatively, we could have used the scan function which has the general form at this level :

```
scan("filename or url", skip, nlines, what)
```

Since scan reads the data as a vector, we can take a different approach in creating a data frame from the file. We cleanse the data as a single vector, then form a data-frame from it :

```
> Jimbo.data.2 = scan("Jimbos.baseball.stats.txt", skip = 1,
+ nlines = 5, what = "");
> Jimbo.data.2 = gsub("[^.0-z]","",Jimbo.data.2);
> my.names = sapply(Jimbo.data.2[1:9], as.character);
> Jimbo.data.2 = Jimbo.data.2[-1:-9]
> Jimbo.data.2 = as.numeric(Jimbo.data.2)
> Jimbo.data.2 = data.frame(matrix(Jimbo.data.2, nrow = 4, byrow = TRUE));
> names(Jimbo.data.2) = my.names
> Jimbo.data.2
```

At this point you should understand what each step does. If you don't , please reread the read.table example and see the section on objects and indexing. As an exercise, we suggest you create your own small file of data and read it into R using the two methods above.

6.4 Missing Values

From time to time you will observe missing values in your data. The examples above are given in part to help you deal with this problem. If the data is not salvageable, then you will have to ommit the missing values from calculations involving *R*. The way to deal with this is to pass the option `na.rm = TRUE` to the function you are using to make your calculation. Further, there is another option to `read.table` not mentioned above called `na.string` that will convert values of a particular pattern to `NA` in character fields. If for some reason you note an alternative to `"NA"` used for data that is not available in a character field of your dataset, then set `na.string` equal to that value.

7 Probability

Distributions make up a very important part of statistics, and R contains a very wide range of them. The following table lists the distributions along their name in *R*.

Distribution	R name
Beta	beta
Binomial	binom
Cauchy	cauchy
Chisquare	chisq
Exponential	exp
F	f
Gamma	gamma
Geometric	geom
Hypergeometric	hyper
Logistic	logis
Lognormal	lnorm
Negative Binomial	nbinom
Normal	norm
Poisson	pois
Student t	t
Uniform	unif
Tukey	tukey
Weibull	weib
Wilcoxon	wilcox

There are a few functions that are common to all distribution objects, given by the following table:

Name	Description
d*name*()	density or probability function
p*name*()	cumulative density function
q*name*()	quantile function
r*name*()	random number generation

We will not be able to cover all of these distributions in this text, but it should be intuitive enough from the following examples to carry over to any of these distributions.

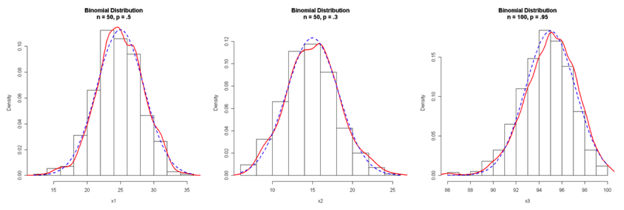
7.1 Binomial

A *binomial(n,p)* distribution is the number of successes in n independent trials where each trial has probability p of success.

```
> n1 = 50; p1 = .5
> n2 = 50; p2 = .3
> n3 = 100; p3 = .95
> x1 = rbinom(1000, n1, p1)
> x2 = rbinom(1000, n2, p2)
> x3 = rbinom(1000, n3, p3)
> hist(x1, probability = T, main = "Binomial Distribution\n
  n = 50, p = .5")
> lines(density(x1), col = "red", lwd = 2)
> curve(dnorm(x, mean = n1 * p1, sd = sqrt(n1 * p1 * (1-p1))),
   add = TRUE, col = "blue", lty = 2, lwd = 2)
```

```
> hist(x2, probability = T, main = "Binomial Distribution\n
   n = 50, p = .3", ylim = c(0,.13))
> lines(density(x2), col = "red", lwd = 2)
> curve(dnorm(x, mean = n2 * p2, sd = sqrt(n2 * p2 * (1-p2))),
   add = TRUE, col = "blue", lty = 2, lwd = 2)
> hist(x3, probability = T, main = "Binomial Distribution\n
   n = 100, p = .95")
> lines(density(x3), col = "red", lwd = 2)
> curve(dnorm(x, mean = n3 * p3, sd = sqrt(n3 * p3 * (1-p3))),
   add = TRUE, col = "blue", lty = 2, lwd = 2)
```

Figure 18: Binomial Histograms



`rbinom` is the function we use to generate 1000 values of the binomial distribution with parameters n and p. As we can see, for large values of n, the binomial distribution can be approximated by the normal distribution.

7.2 Normal Distribution

To work with the normal distribution, first we will show how to plot the standard normal distribution.

```
> x <- seq(from = -4, to = 4, length=100)
> r.dist <- dnorm(x)
> plot(x, r.dist, type = "l", xlab = "x value",
   ylab = "Density", main = "Standard Normal Distribution")
```

The `seq` function returns a sequence of `length` numbers from `from` to `to` in the form of a numeric vector. So, for the example above, x is a vector containing the numbers -4, -3.92, ... , 3.92, 4. The `dnorm` function is simply the likelihood function of the normal distribution. The parameters of the distribution may be specified with additional arguments, such as the `mean` and `sd` (standard deviation). The defaults are mean 0 and a standard deviation of 0.

```
> x <- seq(from = -4, to = 4, length=100)
> r.dist1 <- dnorm(x, mean = 3, sd = 3)
> plot(x, r.dist1, type = "l", xlab = "x value",
   ylab = "Density", main = "Standard Normal Distribution")
```

Figure 19: The Standard Normal :

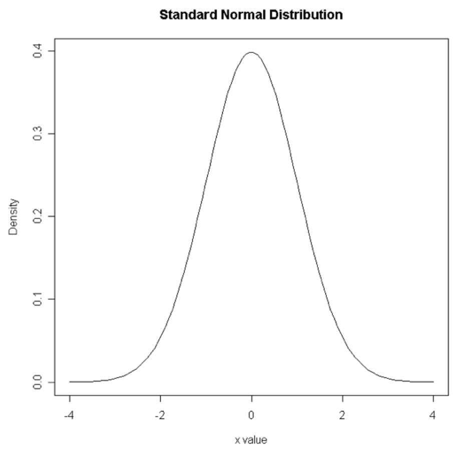
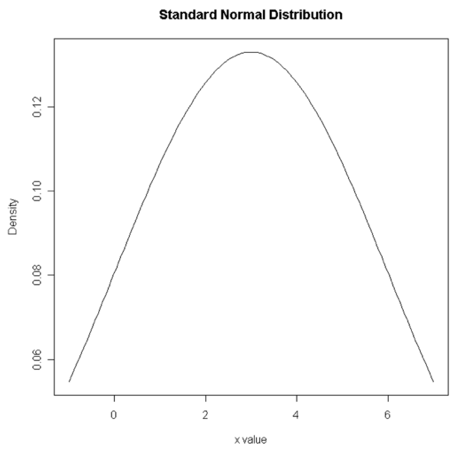


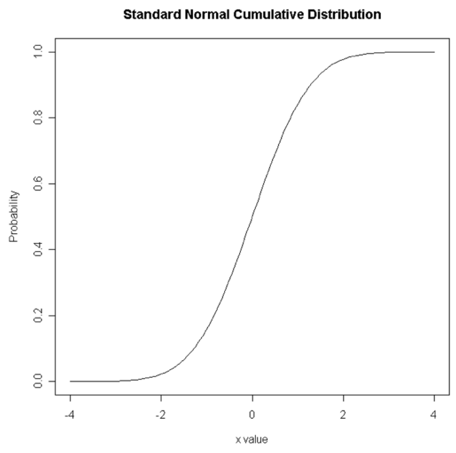
Figure 20: Normal(mean = 3, sd = 3)



We can also plot the cumulative distribution using the `pnorm` function:

```
> x <- seq(from = -4, to = 4, length=100)
> r.cumdist1 <- pnorm(x)
> plot(x, r.cumdist1, type = "l", xlab = "x value",
   ylab = "Probability", main = "Standard Normal Cumulative Distribution")
```

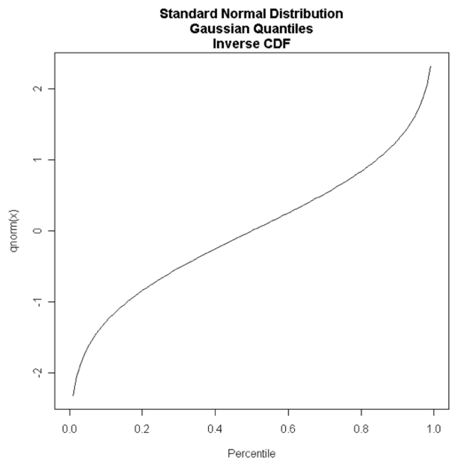
Figure 21: CDF for Normal(0,1)



The quantiles give us the inverse of the cumulative distribution function, and this is given to us by the `qnorm` function. This time, we will plot it using the `curve` function, which takes in an expression written as a function of `x`, and `from` and `to` variables used to specify the min and max x values:

```
> curve(qnorm(x), from = 0, to = 1, main = "Standard Normal
     Distribution\nGaussian Quantiles\nInverse CDF", xlab = "Percentile")
```

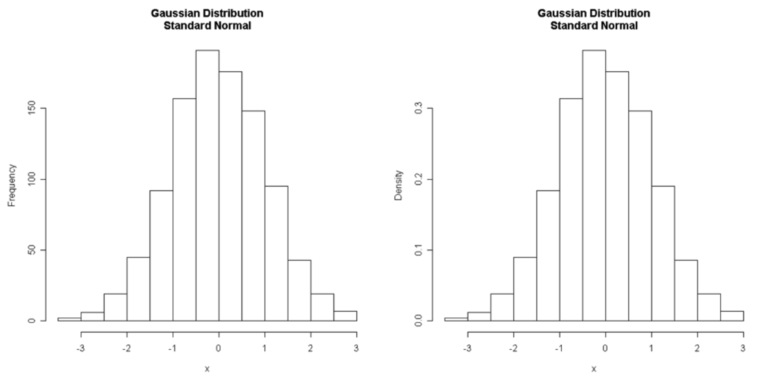
Figure 22: Inverse CDF :



Now we can simulate random variables that take on normal distributions by generating random numbers using the `rnorm` function. In the following examples, we are working with the standard normal distribution, with mean 0 and standard deviation 1:

```
> n = 1000
> x = rnorm(n)
> hist(x, main = "Gaussian Distribution\nStandard Normal")
> hist(x, main = "Gaussian Distribution\nStandard Normal", probability = T)
```

Figure 23: Histograms of Gaussian :



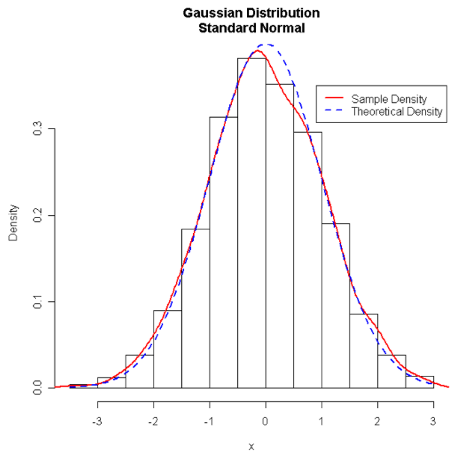
Notice that the graph on the right was created using the `probability` argument, which, when set to `TRUE`, prints the y-axis as probability densities, and not the frequency. Now we can fit a smoothed line for the sample density:

```
> lines(density(x), col = "red", lwd = 2)
```

The `density` function computes kernel density estimates for the numeric vector we provide. Now let's add a dashed line for the theoretical distribution, in addition to a legend:

```
> curve(dnorm(x), add = TRUE, col = "blue", lty = 2, lwd = 2)
> legend(x = .9, y = .35,legend = c("Sample Density",
   "Theoretical Density"), lwd = 2, lty = c(1, 2), col = c("red", "blue"))
```

Figure 24: Theoretical versus Randomly Generated N(0,1)



8 Scripting

You will find that it is useful to have a history of your *R* code from a particular session available for viewing during that session.

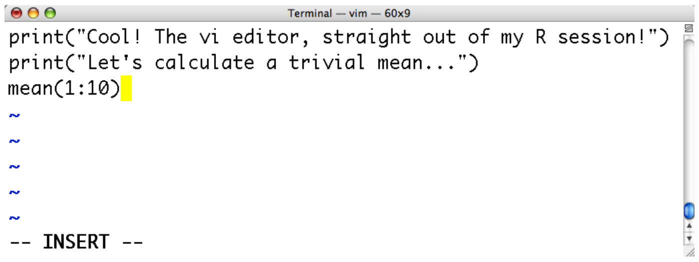
8.0.1 From the Command Line

If you are operating on a Unix system, you may directly open a `vi`, `pico`, or `emacs` editor using those commands, along with the argument `file = "file name"`. Upon saving your file and exiting you will return to your *R* session and the commands from the file will be run immediately. For example

```
> vi(file = "testing.vi.option.R")
```

Will bring us to the `vi` editor. If we entered the following text, and then saved and exited by holding

Figure 25: vi editor :



```
print("Cool! The vi editor, straight out of my R session!")
print("Let's calculate a trivial mean...")
mean(1:10)
~
~
~
~
~
-- INSERT --
```

`shift` and entering `zz`, we would return to the following output in our *R* session

```
> vi(file = "testing.vi.option.R")
[1] "Cool! The vi editor, straight out of my R session!"
[1] "Let’s calculate a trivial mean..."
[1] 5.5
```

8.0.2 Mac : From the GUI

Scroll the pointer across the icons until you reach either the `Create a new, empty document ...` option, or open an existing script from a saved file with the `Open Document in Editor` option, and start scripting. Then copy and paste your script into your *R* session.

8.0.3 Windows : From the GUI

Here you should scroll the pointer to the `File` option, then to the `New Script`, or else `Open Script` option, and commence scripting.

8.0.4 source

Another option is `source`, which reads a file of *R* code into your session, silently. The format is `source(‘‘filename")`. As usual, tab completion holds. For example,

```
> source("testing.vi.option.R")
[1] "Cool!The vi editor, straight out of my R session!"
[1] "Let’s calculate a trivial mean..."
```

Note how the mean of the sequence `1:10` was not output[4].

[4] I need to elaborate on this

9 Functions

9.1 Introduction

R contains built-in functions found in different packages. To see a list of all functions in a particular loaded package, use the `ls` function with the form `ls("package:name of package")`. For example,

```
> library(survival)
> head(ls("package:survival"))
[1] "Surv"       "aml"        "as.date"    "attrassign" "basehaz"
[6] "bladder"
> tail(ls("package:survival"))
[1] "survreg.fit"       "survreg.old"       "tcut"
[4] "tobin"             "untangle.specials" "veteran"
> length(ls("package:survival"))
[1] 77
> ls("package:survival")[77]
[1] "veteran"
```

From time to time you may want to create your own functions. For example, suppose you just didn't like the fact that function `var` calculated the sample variance, and you wanted your own function that calculated the population variance, say `my.var`. Being able to make your own functions means you should never be confined to the built in functions *R* provides for you. The following code shows how to solve the mentioned problem above and compares the built-in function to our own function for the population variance.

```
> my.var = function(vector){
+ sum((vector - mean(vector))^2)/length(vector)
+ }
> my.var(c(1,2,3)) # from a popuation size 3
[1] 0.6666667
> var(c(1,2,3)) # a sample size of 3 from larger population
[1] 1
```

9.2 Functional Form

The general form for creating a function is as follows :

functionName = function($arg_1, arg_2, ..., arg_n, option_1, ..., option_m$*)*{
+command$_1$
+command$_2$
.
.
.
+ *command*$_p$
+ *return(value)*
}

To use the function, the form is :

functionName($arg_1, arg_2, ..., anrg_n, option_1, ..., option_m$*)*

In the first line, calling *function()* tells *R* we want $functionName$ to be a function. arg_i and *option$_i$* are arguments and options, and *command$_i$* are expressions to be evaluated, in the function. The *return(value)* will return *value* when the function is called. If you intend to return more than one object from the function, you will have to make *value* a list containing the relevant objects. If we omit *return(value)*, then the last line evaluated would be returned by default, *command$_p$* above.

As an example, if we wanted to create our own mean function, `our.mean`, we could do the following :

```
>our.mean = function(x){
+  sum(x)/length(x)
+}
```

9.3 Naming Functions

If a function is created that has the same name as a built-in function, then the newly created function will over-ride the built-in. For example, naming a function `q` will make it impossible to cleanly exit your session. To restore the built-in function, use `rm` with the form *rm("function name")*. To take precaution against over-riding the built-in functions, before naming your function use the `exists` command with the form *exists("proposed function name")*. If the name exists already, `TRUE` will be displayed; if not, then `FALSE` will be displayed. The function `get` is similar, with the same form, but it displays the contents of a function, if it exists and an error message otherwise. For example, compare the following:

```
> exists("our.mean")
[1] TRUE
> exists("our.mean.xyz")
[1] FALSE
> get("our.mean")
+   function(x){
+   sum(x)/length(x)
+ }
> get("our.mean.xyz")
Error in get("our.mean.xyz") : variable "our.mean.xyz" was not found
```

You may find `get` expecially useful if you have not been using a text editor to create your functions, and you realize a non syntax error in your function. Calling `get` will allow you to display, then copy and paste the "good" parts of the function into *R*. It is also useful if you have forgotten what the function does, perhaps, because you haven't used it in a while.

9.4 Functions and Loops

More often we will have to either slightly modify or use multiple built-in functions to accomplish some desired result. Typically we have the option of working with loops or with built in functions. It is important to stress that the choice is a matter of taste. Some will feel more comfortable with `for`, `while`, or `repeat` loops. Many times the use of loops can be omitted with the use of functions like `replicate`, `sapply, tapply,` or `mapply`. In the following two sections if you begin to feel bogged down in the complexities of the `for` or `repeat` loops, please skip immediately to the `replicate` section.

9.4.1 for loops

The general form of a for loop is as follows :

for(dummy.variable in sequence){
+command$_1$; command$_2$; ...; command$_n$
+}

First, to understand what follows, the body of the loop is what lies between the { and the} . If the loop is just on one line, such as *for(dummy.variable in sequence) command$_1$*, then *command$_1$* is the body of the loop. What the `for` loop does is execute the commands within its body n times — if n is the length of your sequence — incrementing the dummy variable once on each run through the body. We'll show this in the following example of bootstrapping the difference in means between two vectors:

```
> Boot.mean.oner = function(T.1, T.2){
+   F.P.B = rep(0,1000)
+   F.N.B = rep(0,1000)
+   for(i in 1:1000){
+      if(i %% 100 == 0) print(paste(c("index = ", as.character(i)), collapse = ""))
+      F.P.B[i] = mean(sample(T.1, length(T.1), replace = TRUE))
+      F.N.B[i] = mean(sample(T.2, length(T.2), replace = TRUE))
+      }
+   Diff.oner = F.P.B - F.N.B
+   hist(Diff.oner)
+   return(sort(Diff.oner))
+}
```

Note how the two vectors that were filled by the for loop were initialized before the loop. If you run this function with two vectors of equal length, and at the same time set an object equal to the run function, such as `Boot.run.1 = Boot.mean.oner(r.1, r.2)`, you will see the following wiz by the screen

```
> Boot.run.1 = Boot.mean.oner(r.1, r.2)
[1] "index = 100"
[1] "index = 200"
[1] "index = 300"
[1] "index = 400"
[1] "index = 500"
[1] "index = 600"
[1] "index = 700"
[1] "index = 800"
[1] "index = 900"
[1] "index = 1000"
```

and a histogram will appear. The function, in turn, tells you how many hundreds of times the loop has been run, as soon as the index is set. Note that *after* the 1000th index has been set the histogram appears. This is because the for loop is not finished until the index, `i` in the code, is set to 1000 and the body of the loop on that run is completed. Also note how the function could be simplified as :

```
> Boot.mean.oner = function(T.1, T.2){
+   Diff.oner = rep(0,1000)
+   for(i in 1:1000){
```

```
+     Diff.oner[i] = mean(sample(T.1, length(T.1), replace = TRUE)) -
+     mean(sample(T.2, length(T.2), replace = TRUE))
+   }
+   hist(Diff.oner)
+   return(Diff.oner)
+ }
```

9.4.2 repeat loops

Alternatively, we could use `repeat`, which has the following form :

initialize incremeting variable
repeat{
+ incrementing rule; expression.1; expression.2; ...; expression.n;
+condition to stop repetition
+}

We could do the previous example as follows :

```
> i = 0
> Diff.oner = rep(0, 1000)
> repeat{
+   i = i + 1
+   Diff.oner[i] = mean(sample(T.1, length(T.1), replace = TRUE))
+   - mean(sample(T.2, length(T.2), replace = TRUE))
+   if(i > length(Diff.oner) - 1){
+     i = 0
+     break
+   }
+ }
```

Note that when using `repeat` we *must* initialize our incrementing (or dummy) variable *before* the `repeat` loop, as done in line 1. Line two should be no surprise. The remainder is the body of the `repeat` function. We begin the body with an incrementing rule, basically stating that each time we run through the body of the loop we will increment our counter `i` by one. The second and third lines are straight from the previous example. Lastly,

```
+ if(i > length(Diff.oner) - 1){
+   i = 0
+   break
+ }
```

can be interpreted as `if(i > length(Diff.oner) - 1){` : ”When the incrementor i is greater than 999 execute the commands that are listed before the next }”. `i = 0` is obvious. `break` : ”exit the repeat loop”.

As with the for loops above, we can create a function for this. In so doing we can make the same computations in one line of code :

```
> boot.mean.rep = function(T.1, T.2, n){
+   i = 0
+   Diff.oner = rep(0, n)
+   repeat{
```

```
+     i = i + 1
+     Diff.oner[i] = mean(sample(T.1, length(T.1), replace = TRUE))
+     - mean(sample(T.2, length(T.2), replace = TRUE))
+     if(i > length(Diff.oner) - 1){
+       return(Diff.oner)
+       break
+     }
+   }
+ }
```

Now, `boot.mean.rep(T.1, T.2, 1000)` calculates similar values to `boot.mean.oner(T.1, T.2)`, with slight differences arising from randomness from sampling.

9.4.3 replicate

If the previous examples seemed like a lot of work, then `replicate` is probably the right function for you. The general form is :

replicate(# replications, expression to replicate, option to simplify to vector (or matrix) or a list)

The last argument is `simplify` set to `TRUE` whereby the result is a vector or matrix. Setting `simplify = FALSE` will create a list. Now we illustrate the power of the built-in function. The work done by the previous loops are done by `replicate` as follows :

```
> Diff.oner = replicate(1000, mean(sample(T.1, length(T.1), replace = TRUE))
+ - mean(sample(T.2, length(T.2), replace = TRUE)))
```

That's it! Though it may seem like a dirty trick to put such a gem at the end of the list, we wanted the reader to first experience the alternatives, for appreciation. As a general rule, you should check to see if an operation can be performed by a function like `replicate` or one of the apply functions before creating loops.

9.5 Useful Functions :

9.5.1 Uploading a set of Packages

Suppose you wanted to load a certain set of installed packages at the beginning of your *R* sessions. The following function solves the mentioned problem.

```
>.packages(all.available = TRUE) # for matching
>#create vector, vector, containing the locations of entries you want
>package.loader = function(vector)
+  for(i in vector){
+    library(.packages(all.available = TRUE)[i], character.only = TRUE)
+}
```

9.5.2 Adjusting the QQ-line for Comparisons with Theoretical Distributions

To plot QQ-lines for non-standard normal distributions, we have to modify the built in `qqline`. To figure out how to create qqlines that meet our needs, we examine the `qqline` function.

```
> qqline
function (y, datax = FALSE, ...)
{
    y <- quantile(y[!is.na(y)], c(0.25, 0.75))
    x <- qnorm(c(0.25, 0.75))
    if (datax) {
        slope <- diff(x)/diff(y)
        int <- x[1] - slope * y[1]
    }
    else {
        slope <- diff(y)/diff(x)
        int <- y[1] - slope * x[1]
    }
    abline(int, slope, ...)
}
```

We see that our needs can be met by adjusting the line `x <- qnorm(c(0.25, 0.75))`, in creating our own slightly modified qqline function. We know that `rexp(1000, .5)` is approximately the theoretical *exponential*$(\lambda = .5)$. We can make an altered qqline to compare D.set to the theoretical *exponential*$(\lambda = 0.5)$ as follows (simply changing lines 1 and 3 of `qqline`)

```
> exp.qqline = function (y, datax = TRUE, rate, ...)
+ {
+     y <- quantile(y[!is.na(y)], c(0.25, 0.75))
+
+         x <- qexp(c(0.25, 0.75), rate = rate)
+     if (datax) {
+         slope <- diff(x)/diff(y)
+         int <- x[1] - slope * y[1]
+     }
+     else {
+         slope <- diff(y)/diff(x)
+         int <- y[1] - slope * x[1]
+     }
+     abline(int, slope, ...)
+ }
```

Now, we can pass `D.set` to `exp.qqline` and get an approximate qqline. All we did is replace the normal quantile function `qnorm` with the exponential quantile function `qexp` and allowed for the passing of a rate of decay `rate` to the `qexp`.

```
> exp.qqline(D.set, rate = .5)
```

Similarly, we could create an adjustable `qqline` function for theoretical Normal *means* $\neq 0$ and *sd* $\neq 1$. That is, by changing the first line to `FunctionName = function(y, datax = TRUE, mean, sd){` and the third line to `x<-qnorm(c(0.25, 0.75), mean, sd)` we could achieve the mentioned goal — as would similar appropriate changes work for other distributions. *See the section on* `Functions` *for* `m.qqline` *which will graph a* `qqline` *for any bulit in density function.*

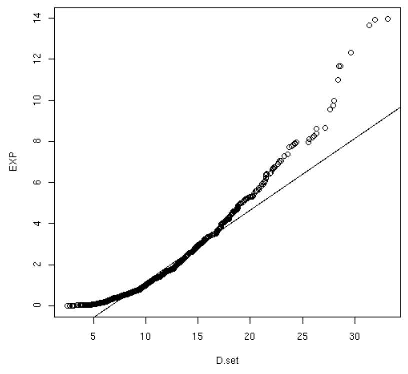
9.5.3 adjusting qqline for two sample comparison

Lastly, we could compare the quantiles of the actual data by adding `x` to the arguments and replacing the line `x <- qnorm(c(0.25, 0.75))` with `x <- quantile(x[!is.na(x)], c(0.25, 0.75))`.

9.5.4 A More Adjustable QQ-line :

The following function will plot a qqline for comparison of a vector with any built in `base` package density function in *R*. The first argument `y` is the vector to compare with the density function. The

Figure 26: approximated qqline by using `exp.qqline`



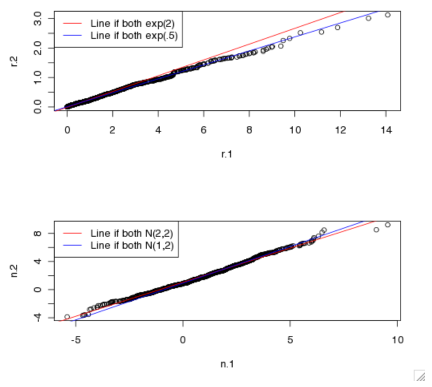
second argument `datax` should be `True` if this vector is on the x axis of the existing plot, otherwise `False`. The third argument `fcn` is the name of the quantile density for comparison, in quotes — for example `fcn = "qexp"`. See the vector `fcn.list` for possible entries.

```
> m.qqline = function(y, datax, fcn, pList, ...){
+   exit = 0
+   y = quantile(y[!is.na(y)], c(0.25, 0.75))
+   fcn.list.f = c(qexp, qnorm, qbeta, qbinom, qcauchy, qf, qchisq, qgamma,
+     qgeom, qhyper, qlogis, qlnorm, qnbinom, qpois, qt, qunif, qtukey,
+     qweibull, qwilcox)
+   fcn.list = c("qexp", "qnorm", "qbeta", "qbinom", "qcauchy", "qf",
+     "qchisq","qgamma", "qgeom", "qhyper", "qlogis", "qlnorm", "qnbinom",
+     "qpois","qt", "qunif", "qtukey", "qweibull", "qwilcox")
+   fcn.par.l = c(1, 2, 2, 2, 2, 3, 1, 2, 1, 3, 2, 2, 3, 1, 2, 2, 3, 2, 2)
+   exit = 0
+   j = 1
+     repeat{
+     if((fcn == fcn.list[j]) && (length(pList) == fcn.par.l[j])){
+     if(length(pList) == 1){
+       fcn.list.f[j][[1]](c(0.25, 0.75), pList[1])->x
+       exit = 1} else
+     if(length(pList) == 2){
+       fcn.list.f[j][[1]](c(0.25, 0.75), pList[1], pList[2])->x
+       exit = 1} else
+    if(length(pList) == 3){
+       fcn.list.f[j][[1]](c(0.25, 0.75), pList[1], pList[2], pList[3])->x
+       exit = 1}
+  }
+    if(exit == 1){
+  if(datax){
+    slope = diff(x)/diff(y)
+    int = x[1] - slope*y[1]
+  }
+  else {
+    slope = diff(y)/diff(x)
+    int = y[1] - slope*x[1]
+  }
+  abline(int, slope, ...)
+  break}
+  if(j > length(fcn.list.f)){"You have entered too many or too few parameters
+    for the function, or an invalid function name for fcn"
+  break
+  }
+  j = j+1
+}
+}
```

For example :

```
> r.2 = rexp(1000, 2)
> r.1 = rexp(1000, .5)
>
> par(mfrow = c(2, 1))
> qqplot(r.1, r.2)
> m.qqline(r.1, datax = TRUE, "qexp", 2, col = 2)
> m.qqline(r.2, datax = FALSE, "qexp", .5, col = 4)
> legend(legend = c("Line if both exp(2)", "Line if both exp(.5)"),
> col = c(2, 4), x = "topleft", lty = 1)
>
> n.1 = rnorm(1000, 1, 2)
> n.2 = rnorm(1000, 2, 2)
> qqplot(n.1, n.2)
> m.qqline(n.1, datax = TRUE, "qnorm", c(2,2), col = 2)
> m.qqline(n.2, datax = FALSE, "qnorm", c(1,2), col = 4)
> legend(legend = c("Line if both N(2,2)", "Line if both N(1,2)"),
> col = c(2, 4), x = "topleft", lty = 1)
```

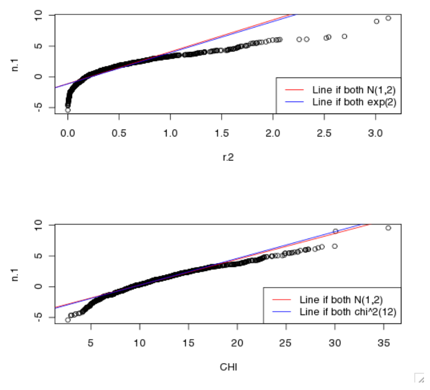
Figure 27: Results from the `m.qqplot` :



To compare exponentials and normals and χ^2:

```
> qqplot(r.2, n.1)
> m.qqline(r.2, datax = TRUE, "qnorm", c(1,2), col = 2)
> m.qqline(n.1, datax = FALSE, "qexp", 2, col = 4)
> legend(legend = c("Line if both N(1,2)", "Line if both exp(2)")
+ , col = c(2, 4), x = "bottomright", lty = 1)
>
> CHI = rchisq(1000, 12)
>
> qqplot(CHI, n.1)
> m.qqline(CHI, datax = TRUE, "qnorm", c(1,2), col = 2)
> m.qqline(n.1, datax = FALSE, "qchisq", 12, col = 4)
> legend(legend = c("Line if both N(1,2)", "Line if both chi^2(12)"),
+ col = c(2, 4), x = "bottomright", lty = 1)
```

Figure 28: Results from the `m.qqplot` :



10 Acknowledgments

10.1 John Jimenez

I would like to thank professors Ani Adhikari, John Rice, and Phil Spector. The idea in itself of having two undergraduates, freshly trained in R, write a tutorial on the language for beginners is brilliant in one sense, but risky in another. I thank them for trusting in our abilities and allowing me to be a part of The Berkeley R Project. Every step of the process was imbued with many degrees of freedom to cover the topics Devon and I felt necessary for a learner of R. Nonetheless, their availability for guidance and expertise has been crucial throughout, especially in the editing process. Meeting discussions indirectly provided material for some of the examples and a better understanding of the field of statistics. I am grateful to Professor Spector for teaching me, and sparking my interest in, R. Lastly, I thank the many creators of the R help pages, for making my life a lot easier, not just while creating this tutorial, but in everyday computing.

10.2 Devon Shurick