Really Introductory Introduction to ℝ

Thomas J. Leeper
Department of Political Science
Northwestern University

August 29, 2012

Abstract

The purpose of this document is to provide a brief introduction to ℝ, focused on the core code needed to load and analyze social scientific data. While numerous other resources exist online to facilitate learning ℝ and to provide guidance on performing particular procedures, this document serves as a quick-start guide catered to the needs of students who are learning ℝ for the first time in the context of a course that provides little technical instruction in programming. My aim in this document is to make learning ℝ relatively more smooth so that you focus more of your attention on learning statistics and political science research methods and less on being frustrated with the particular software we are using to implement those ideas.[1]

1 Why R?

ℝ refers to a statistical computing environment and the programming language used therein. We use ℝ in this course because it is free, open source, well-documented, involves a large community of active users, and, most importantly, because it provides nearly unlimited functionality that far exceeds that of all other statistical packages.

The major drawback of using ℝ is that it is difficult for new users to learn. If you have no background in programming, the learning curve can be very steep. Prior experience with statistical theory is often not terribly helpful for actually implementing that theory in ℝ. Prior experience with other statistical packages (Stata, SAS, SPSS, Matlab) is somewhat helpful and several texts have been written to help users of those packages transition to ℝ. If you have prior experience with programming in other languages, learning ℝ will be relatively straightforward; BASIC, C#, PHP, Python, Javascript, etc. all share certain similarities with ℝ.

A second challenge is that not a lot of effort has been put into making ℝ user-friendly for those with no programming experience. It is a command-based program. There is almost nothing to "point-and-click" anywhere in ℝ. You need to learn to program in order to even use ℝ. When you do something wrong in ℝ, it will give you an error message. These errors

[1] Some of this material is adapted from Teppei Yamamoto's (2009) "Introduction to ℝ" Short Course. It is released without copyright, but citation is always appreciated.

are typically cryptic and unhelpful. Trial-and-error, use of the online help files, and use of Google will all be necessary. Your instructor and TA should not be your first resources for problems with ℝ. There is a wealth of information on the internet about ℝ and you should plan on using your classmates as resources for troubleshooting. I am here to help you when you have exhausted these other resources. **I expect you to spend at least 20 minutes trying to solve a problem and have thoroughly read the relevant help files before emailing me.**

So why use ℝ if it is so difficult to learn and so different other statistical packages that are also widely used and with which you might already be familiar? The payoff of learning can be substantial. Because ℝ is a programming language, it allows you an incredible amount of flexibility for the manipulation of data, the creation of unique functions for analyzing and describing those data, and unparalleled functionality for the tabular and graphical presentation of data. For these reasons, ℝ can actually be fun, once you get to know it. Being competent in ℝ will make learning other programming languages and other statistical packages much easier — skills that may prove useful and/or necessary in your future careers. And, once you know how to program and do statistical analysis, people will think you're really intelligent even if you aren't, which has its advantages on the job market.

So, let's get started with ℝ.

2 Getting Started

The first step is downloading and installing ℝ. This is straightforward and instructions can be found on the ℝ homepage `http://www.r-project.org`. ℝ runs on all major operating systems in effectively the exact same fashion (yet another advantage of ℝ!). Once you have ℝ installed, you can open it and you'll be presented with a command line. This is where you tell ℝ what you want to do. Because you don't know how to program any ℝ code yet, try using ℝ as a calculator. The standard mathematical operators all work, and follow the 'order of operations.' Here are some simple examples:

```
> 2+2

[1] 4

> 51-38

[1] 13

> (2+4)*6

[1] 36

> 20/4

[1] 5

> 2^3

[1] 8

> sqrt(4)
```

```
[1] 2
```

But, ℝ is more than just a calculator (obviously!). To move away from using ℝ as nothing more than a bloated calculator, we need to start doing some actual programming. But before we start writing more complex code, you'll probably want to move away from typing code directly into ℝ and instead use another program to help you keep track of your code. There are a number of programs that can facilitate programming in ℝ and dealing with its text and graphical outputs. First, you need a text editor to save the code you write. ℝ doesn't automatically save the code that you tell it, so you'll save yourself a lot of time if you save all of your code in a separate program. You can use simple programs like Notepad, Wordpad, or even Word to do this, but better programs exist like TINN-R or Notepad++ (what I use), Emacs, or RStudio.

- Simple text editors: Notepad, TextPad, TextEdit, Wordpad (usually one or more of these is built-in with your computer's operating system)
- Fancier editors: **TINN-R, WinEdt, Emacs, Aquamacs**, Notepad++
- Advanced front-ends for ℝ: **RStudio, Revolution R, JGR, Deducer**
- Graphical User Interface for ℝ: **Rcmdr, RKWard**

If you save all of your code in one of these programs with a `.R` file extension, you should be able to completely replicate your analysis simply by opening this file with ℝ or copying and pasting the entirety of the code into ℝ. This is helpful if you ask someone for help with your code because they can run all of your code and see where it may have had problems.

2.1 Some General Notes Before Getting Started

- ℝ is an object-oriented programming language. That means when we talk about particular statistics or procedures, we have to write code that tells ℝ how to perform that statistical *procedure* on a *data object*. Usually, this looks something like `mean(x)`, where `mean()` is a procedure and `x` is an object (specifically, a vector of data points). At first, it can be hard to distinguish what text is a procedure from what text is the label for a data object, but over time this will get easier. If you use a text editor with syntax recognition, the text editor will highlight procedures making the process of reading and writing code even easier.
- ℝ is case-sensitive. You need to type commands and variable names correctly or they will not work. As an example, `Mean()` does not mean `mean()`.
- Arrow up and Arrow down on your keyboard cycle through previous commands you have entered. They do not scroll the screen up and down. Use your mouse scrollpad, Page Up and Page Down, or the scrollbar instead.
- If you forget to put closing brackets on your commands, ℝ will get confused and give you a `+` symbol instead of the usual `>` prompt. You can simply type the closing bracket in and the command will execute as if you typed it correctly the first time. This also means that ℝ doesn't really recognize linebreaks. If you have a very long command (like for plots, which we'll talk about later), you may want to put the command on multiple lines so it is easier to read and ℝ will have no problem with this, as long as you put the appropriate closing bracket at the end. If you use Mac OS or Linux OS, or

some of the fancier text editors, closing brackets are often put in automatically as you type. If you're a Windows user like me, you're out of luck and need to be really careful with brackets.

- Sometimes ℝ does weird stuff that doesn't make any sense. When this happens, double check your brackets and capitalization. Then, trying Googling it. If that doesn't work, you can always just restart the program.

2.2 Creating constants, vectors, matrices, and dataframes

To do statistical analysis, we need to have data. ℝ has various objects for storing different types of data. The simplest way to store data is in a *constant.* A constant, to ℝ, is just a number that is given a name. We use the `<-` to store something into a particular labeled object; we can also use `=` to do this; or `->` by reversing the argument:

```
> a<-4
> a=4
> 4->a
> a

[1] 4
```

Here, we've created a constant called `a`, which has the value 4. ℝ doesn't automatically display the value of `a` when we assign a value to it. To see the value of `a`, you need to simply type `a` and hit <enter>. We can store multiple constants, give new values to a constant we've already defined, and then conduct mathematical operations on those constants:

```
> a=4
> a=6
> a

[1] 6

> b=3
> b+2

[1] 5

> b*a

[1] 18
```

Note that $b * a = 18$ in the code above, even though we calculated `b+2` in the line directly above that calculation. You have to store a result as a constant to be able to use that result later.

Storing all of our data as individual constants, isn't very efficient, though. Let's say we want to collect a piece of data for a number of units (e.g., people in this class). We can store a set of data points as a *vector* in ℝ using the `c()` command. Vectors can have any length and we can perform mathematical operations on vectors.

```
> a=c(1,2,3,4,5,6,7,8,9,10)
> a
```

```
 [1]  1  2  3  4  5  6  7  8  9 10

> a*2

 [1]  2  4  6  8 10 12 14 16 18 20

> b=a
> b

 [1]  1  2  3  4  5  6  7  8  9 10
```

You can also create a sequence of numbers using `seq()` or just the `:`.

```
> a=seq(1:10)
> a

 [1]  1  2  3  4  5  6  7  8  9 10

> b=1:10
> b

 [1]  1  2  3  4  5  6  7  8  9 10
```

This is especially helpful if we want to calculate any basic statistics on a particular vector of data (what we might, in social science language, call a variable). Let's imagine for instance that we collect a number of 2-dice rolls and record the sum in a vector called dice. We can then find out a number of pieces of information about this set of rolls.

```
> dice=c(2,2,3,4,4,5,5,5,5,5,6,6,7,7,7,7,8,9,10,11)
> table(dice) #a simple tabulation of how many of each value are in the vector

dice
 2  3  4  5  6  7  8  9 10 11
 2  1  2  5  2  4  1  1  1  1

> fivenum(dice) #the five number summary (min, Q1, median, Q3, max) for the vector

[1]  2.0  4.5  5.5  7.0 11.0

> quantile(dice, c(0.1,0.2,0.6,0.8)) #arbitrary quantiles

10% 20% 60% 80%
2.9 4.0 6.4 7.2

> summary(dice) #a set of basic univariate statistics for the vector

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2.00    4.75    5.50    5.90    7.00   11.00
```

Note we can also obtain these summary values individually.

```
> dice=c(2,2,3,4,4,5,5,5,5,5,6,6,7,7,7,7,8,9,10,11)
> length(dice) #how many observations are there?

[1] 20
```

```
> min(dice) #minimum

[1] 2

> median(dice) #median

[1] 5.5

> max(dice) #maximum

[1] 11

> max(dice)-min(dice) #range

[1] 9

> sum(dice)

[1] 118

> sum(dice)/length(dice)

[1] 5.9

> mean(dice) #mean

[1] 5.9

> v=var(dice) #variance
> v

[1] 5.884211

> sqrt(v) #standard deviation

[1] 2.425739

> sd(dice) #easier way of getting standard deviation

[1] 2.425739
```

One thing that can be initially confusing for new users of ℝ is exactly what is happening when you type in a given command. ℝ does not solve mathematical equations. Instead, like any programming language, it performs a set of procedures in a specific order on a particular data object (i.e., it performs an algorithm). What does an algorithm look like and why does it matter? To use ℝ effectively, one needs to understand what it is doing and how to think in algorithms rather than equations. Think about the formula for the mean of a set of data (like our dice roll data). The mathematical equation is $\bar{x} = \Sigma x_i/n$. This translates into an ℝ algorithm like the following:

```
> dice=c(2,2,3,4,4,5,5,5,5,5,6,6,7,7,7,7,8,9,10,11)
> sum=2+2+3+4+4+5+5+5+5+5+6+6+7+7+7+7+8+9+10+11
> sum

[1] 118
```

```
> length=length(dice)
> length
```

```
[1] 20
```

```
> mean=sum/length
> mean
```

```
[1] 5.9
```

As we can see, the result is the same as using ℝ's built-in `mean()` function. We could attempt something similar by converting the equation for variance ($Var[x] = \Sigma(x_i - \bar{x})^2/(n-1)$) into an algorithm (the first four steps of which are the same as calculating the mean):

1. Load the vector of all x_i, called *data*
2. Sum all x_i, store this result as *sum*
3. Count the number of x_i, store this as *length*
4. Divide *sum* by *length*, store this as *mean*
5. Create a new vector, where each entry in the vector is the difference between each x_i in *data* and mean of all x_i (*mean*), store as *deviations*
6. Create a new vector, where each entry is the square of the entry in *deviations*, store as *squares*
7. Sum all values in *squares*, store as *sumsquares*
8. Divide *sumsquares* by one less than *length*, store as *variance*

We can then write that algorithm in ℝ:

```
> dice=c(2,2,3,4,4,5,5,5,5,5,6,6,7,7,7,7,8,9,10,11)
> sum=sum(dice)
> length=length(dice)
> mean=sum/length
> deviations=dice-mean
> squares=deviations^2
> sumsquares=sum(squares)
> variance=sumsquares/(length-1)
> variance
```

```
[1] 5.884211
```

```
> sqrt(variance)
```

```
[1] 2.425739
```

As we can see, our mathematical equation for variance is easily converted to an algorithm that we can then write in ℝ code. While the `var()` command is short and simple, it reflects all of this underlying code. While we can rely on these built-in functions, it is important to keep in mind what is happening behind the scenes. Thinking like a computer programmer rather than a math student will greatly facilitate your transition to ℝ.

Two additional important tips here. First, you can use the `round()` command to round a given value to a specified number of decimal places. Second, you can nest commands inside of one another rather than produce a constant that you'll only use one time (e.g., to calculate the standard deviation we can create a constant that stores the variance and take the square root thereof OR we can simply take the square root of the variance directly).

```
> round(mean(dice),2)

[1] 5.9

> round(sqrt(var(dice)),2)

[1] 2.43
```

We can also extract one or more observations from a vector. For instance, if we wanted to know the value of the 1st through 3rd rolls, or just the value of the 7th roll:

```
> dice=c(2,2,3,4,4,5,5,5,5,5,6,6,7,7,7,7,8,9,10,11)
> dice[1:3]

[1] 2 2 3

> dice[7]

[1] 5
```

We'll discuss more about extracting particular observations in a moment. Clearly, though, conducting simple analyses in R is pretty straightfoward. Most of the commands are semantic — that is, the command to obtain a particular statistic is intuitive. This is generally the case, but can get confusing once we move into more advanced statistics. Using comments in your R programming will therefore be vital to you and I both understanding what you're trying to do with a given piece of code. The pound sign (#) can be used to create comments in the code, which can be helpful when you have to do complex things in your code and you want to remind yourself (or me) what you're trying to do with each bit of code. They can be included at the end of a line of code or simply on their own. Just start the line with a pound sign and then continue with code on the next line.

Now let's work with two variables. Let's try comparing two sets of dice rolls, by looking at their means, variances, and the correlation between the two sets of rolls.

```
> dice1=c(7,7,4,5,3,6,8,2,12,10)
> dice2=c(8,4,7,7,7,7,6,6,3,4)
> mean(dice1)

[1] 6.4

> sd(dice1)

[1] 3.098387

> mean(dice2)

[1] 5.9

> sd(dice2)

[1] 1.66333
```

```
> cor(dice1,dice2)
```

```
[1] -0.6812877
```

We could also represent these data a different way. Rather than having two vectors, we could "column bind" them (using the `cbind()` command) into a *matrix*, which we can also call whatever we want.

```
> dice1=c(7,7,4,5,3,6,8,2,12,10)
> dice2=c(8,4,7,7,7,7,6,6,3,4)
> dice=c(dice1,dice2)
> set=c(1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2)
> dice.mat=cbind(set,dice)
> dice.mat
```

```
      set dice
 [1,]   1    7
 [2,]   1    7
 [3,]   1    4
 [4,]   1    5
 [5,]   1    3
 [6,]   1    6
 [7,]   1    8
 [8,]   1    2
 [9,]   1   12
[10,]   1   10
[11,]   2    8
[12,]   2    4
[13,]   2    7
[14,]   2    7
[15,]   2    7
[16,]   2    7
[17,]   2    6
[18,]   2    6
[19,]   2    3
[20,]   2    4
```

We can also use the corresponding `rbind()` to "row bind" two or more vectors, which has the same effect as `cbind()`, but treats each vector as a row/observation. The `t()` command transposes a matrix. If we apply this function to our original matrix (that we constructed using `cbind()`), we will obtain the second matrix we constructed (using `rbind()`).

```
> rbind(set,dice)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14]
set     1    1    1    1    1    1    1    1    1     1     2     2     2     2
dice    7    7    4    5    3    6    8    2   12    10     8     4     7     7
     [,15] [,16] [,17] [,18] [,19] [,20]
```

```
set      2     2     2     2     2     2
dice     7     7     6     6     3     4
```

```
> t(dice.mat)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14]
set     1    1    1    1    1    1    1    1    1     1     2     2     2     2
dice    7    7    4    5    3    6    8    2   12    10     8     4     7     7
     [,15] [,16] [,17] [,18] [,19] [,20]
set      2     2     2     2     2     2
dice     7     7     6     6     3     4
```

Here, we created our vectors from above, but then combined them into a single vector using the `c()` command. We then created a separate vector called `set` into which we stored the identifier for whether the observation came from our first or second set of rolls. We then bound these two vectors together into a matrix called `dice.mat`, which has dimensions 20 x 2. Now this is starting to look like a traditional rectangular dataset where each observation is represented as a row and each variable is a column. We can look at the dimensions of the new matrix, including its number of columns and number of rows and use that information to extract particular values (or sets of values) from the matrix. We can extract a particular observation (e.g., observation 5), a particular variable (e.g., the second variable), or the observation for a given variable (e.g., the value of the roll for our fifth roll) just as we did with vectors, but now our syntax reflects that row and column position, as follows:

```
> ncol(dice.mat)

[1] 2

> nrow(dice.mat)

[1] 20

> dim(dice.mat) #obtain both dimensions

[1] 20  2

> dice.mat[5,]

 set dice
   1    3

> dice.mat[,2]

 [1]  7  7  4  5  3  6  8  2 12 10  8  4  7  7  7  7  6  6  3  4

> dice.mat[5,2]

dice
   3
```

We could continue to bind vectors to this matrix. For example, we could bind a vector to the matrix that identifies the color the dice we rolled. Here we'll be dealing with a vector of *strings*, which we cannot perform statistical or mathematical operations, but that we often find in real world data.

```
> color=c("red","red","blue","yellow","blue","red","red","blue","yellow","blue","red",
> dice.mat2=cbind(dice.mat,color)
> dice.mat2
```

```
      set dice color
 [1,] "1" "7"  "red"
 [2,] "1" "7"  "red"
 [3,] "1" "4"  "blue"
 [4,] "1" "5"  "yellow"
 [5,] "1" "3"  "blue"
 [6,] "1" "6"  "red"
 [7,] "1" "8"  "red"
 [8,] "1" "2"  "blue"
 [9,] "1" "12" "yellow"
[10,] "1" "10" "blue"
[11,] "2" "8"  "red"
[12,] "2" "4"  "red"
[13,] "2" "7"  "blue"
[14,] "2" "7"  "yellow"
[15,] "2" "7"  "blue"
[16,] "2" "7"  "red"
[17,] "2" "6"  "red"
[18,] "2" "6"  "blue"
[19,] "2" "3"  "yellow"
[20,] "2" "4"  "blue"
```

We can also transform this matrix into a different type of $\mathbb{R}$ object called a *dataframe*. Matrices and dataframes have some similar properties, but also differences that you will learn over the course of this course. Importantly, most data that we read in from an outside file (as opposed to typing in ourselves) will be in the form of a dataframe. But, we can also transform our matrix into a dataframe. If you want to know whether something is a matrix or a dataframe, you can also test it using an `is.matrix()` or `is.data.frame()` command.

```
> dataframe=as.data.frame(dice.mat2)
> is.data.frame(dataframe)
```

```
[1] TRUE
```

```
> is.matrix(dataframe)
```

```
[1] FALSE
```

```
> is.data.frame(dice.mat2)
```

```
[1] FALSE
```

```
> is.matrix(dice.mat2)
```

```
[1] TRUE
```

One of the key advantages of dataframes is that we can refer to columns of the dataframe by their variable names, as opposed to their position number (as we have to do with matrices). We can find these names using the `names()` command and isolate a particular column of the matrix by using the dollar sign operator. Note that this produces the same result as using the brackets to identify the particular column (as we did above with our matrix).

```
> names(dataframe)

[1] "set"   "dice"  "color"

> dataframe$color

 [1] red    red    blue   yellow blue   red    red    blue   yellow blue
[11] red    red    blue   yellow blue   red    red    blue   yellow blue
Levels: blue red yellow

> dataframe[,3]

 [1] red    red    blue   yellow blue   red    red    blue   yellow blue
[11] red    red    blue   yellow blue   red    red    blue   yellow blue
Levels: blue red yellow
```

We can also extract a column or row from a dataframe and turn it into a vector. We might want to do this if we plan on transforming the data in some way and saving that transformed data for later use. We could also add that new vector back into the dataframe under a new name. The vector and column in the dataframe will have the same properties, but changing one has no effect on the other. This aspect can sometimes get confusing, especially once you start working with data that contains many variables. You can have a column in a dataframe and a separate vector that have the same name. To work with the vector simply use the name, to use the column in the dataframe, you have to use the $ operator.

```
> df=as.data.frame(dice.mat)
> df$dice

 [1]  7  7  4  5  3  6  8  2 12 10  8  4  7  7  7  7  6  6  3  4

> half=df$dice+1
> df$half=half
> summary(half)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   3.00    5.00    7.50    7.15    8.00   13.00 

> summary(df$half)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   3.00    5.00    7.50    7.15    8.00   13.00 

> half=half+1
> summary(half)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   4.00    6.00    8.50    8.15    9.00   14.00 
```

```
> summary(df$half)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   3.00    5.00    7.50    7.15    8.00   13.00 
```

Be careful here, because you can also accidentally change a vector by giving something else the same name. For example, we can eliminate the vector entirely by assigning a constant value to it or by giving it the missing value code `NA`.

```
> half
```

```
 [1]  9  9  6  7  5  8 10  4 14 12 10  6  9  9  9  9  8  8  5  6
```

```
> half=1
> half
```

```
[1] 1
```

```
> half=NA
> half
```

```
[1] NA
```

We can apply the same `summary()` and `table()` commands that we used on vectors to matrices and dataframes. We can also use `table()` on two variables to construct crosstabs.

```
> summary(df)
```

```
      set           dice            half      
 Min.   :1.0   Min.   : 2.00   Min.   : 3.00  
 1st Qu.:1.0   1st Qu.: 4.00   1st Qu.: 5.00  
 Median :1.5   Median : 6.50   Median : 7.50  
 Mean   :1.5   Mean   : 6.15   Mean   : 7.15  
 3rd Qu.:2.0   3rd Qu.: 7.00   3rd Qu.: 8.00  
 Max.   :2.0   Max.   :12.00   Max.   :13.00  
```

```
> table(df$dice,df$set) #crosstab
```

```
     1 2
  2  1 0
  3  1 1
  4  1 2
  5  1 0
  6  1 2
  7  2 4
  8  1 1
  10 1 0
  12 1 0
```

```
> with(df,table(dice,set)) #an alternative method for getting crosstabs

    set
dice 1 2
  2  1 0
  3  1 1
  4  1 2
  5  1 0
  6  1 2
  7  2 4
  8  1 1
  10 1 0
  12 1 0
```

```
> prop.table(table(df$dice,df$set)) #convert cell counts to proportions

        1    2
  2  0.05 0.00
  3  0.05 0.05
  4  0.05 0.10
  5  0.05 0.00
  6  0.05 0.10
  7  0.10 0.20
  8  0.05 0.05
  10 0.05 0.00
  12 0.05 0.00
```

Sometimes you want to know all the objects you have created. To see these, use the ls() command.

```
> ls()

 [1] "a"          "b"          "color"      "dataframe"  "deviations"
 [6] "df"         "dice"       "dice.mat"   "dice.mat2"  "dice1"
[11] "dice2"      "filename"   "half"       "length"     "mean"
[16] "set"        "squares"    "sum"        "sumsquares" "v"
[21] "variance"   "wd"         "x"
```

You can also use rm(list=ls(all=TRUE)) to clear all objects; this effectively starts a new ℝ instance because you will lose all of your work.

2.3 ℝ Output

Just as it is helpful to save your ℝ code in an outside file, it is also sometimes helpful to send the output of ℝ to an outside file so that you can retrieve it later. ℝ provides the sink() command to allow you to put all of the output it produces into an outside .txt file in addition to (or instead of) sending the output to the ℝ console. This can be helpful when you start performing a large number of analyses. (You need to enter the sink() command, complete with all its arguments before conducting your analyses.)

3 Working with Real Data

Now that we've worked with some of the basic manipulations of data and the type of objects that ℝ understands (constants, vectors, matrices, and dataframes), we can proceed forward by working with actual data. It is rare that we'll have to enter our own data manually into ℝ. Instead, we'll likely have to import it from an outside file of various types. To this, we have to start by telling ℝ where to find the data. You can either do this visually by selecting `File > change dir`, or you can use the `setwd()` command.[2] This sets the working directory for your current ℝ session. You can change it later as needed, but you'll need to set it each time you work with ℝ.

```
> setwd("C:\\Users\\Thomas\\Documents\\Dropbox\\Professional Development\\Syllabi\\Int
```

You can check what your working directory is by using the corresponding **getwd()** command.

After your working directory is set, you can read data into ℝ. How you do this will depend on the type of data file you are working with:

- Comma-Separated Variable (.csv) data: use `read.csv()`
- Table-Delimited Data (.tab): use `read.delim()`
- Fixed-width format data (file extensions vary): use `read.fwf()` or `read.table()`
- Stata (.dta) data: use `read.dta()` in the `foreign` package
- SPSS (.sav, .por) data: use `read.spss()` in the `foreign` package

Sometimes you'll encounter data in other formats, but these are the most common. The first three formats can be read using commands that are built-in to ℝ. The latter two, for Stata and SPSS filetypes, require an outside package called `foreign`.

3.1 Packages

Installing packages is straightforward. You can do this in many ways, but I'll describe two of them quickly. The first method involves finding the package on the ℝ website (`http://www.r-project.org`, downloading the compressed (.zip) folder for the package (usually there are different versions for each operating system), unpacking that folder, and saving it into the "library" folder in ℝ's system files. This can be confusing if you're not used to saving things in your system folders. The second, easier, way is to select `Packages > Install package(s)` from within ℝ. You will then be prompted to select a "CRAN mirror"; you can select any of these. You will next be prompted to select the package you want to install. Selecting it from the list and clicking OK will download, unpack, and install the package. It is now ready to use.

The first method is usually foolproof, but the second can sometimes fail inexplicably. Often this has to do with file permission settings or internet access on your computer. For example, in Windows, you will likely have to run ℝ "as an administrator" in order for the package to download and install correctly.

[2] The file path you use will depend on your operating system and the folder where you have saved the data file on your computer. See the ℝ help files for how to do this. You can access the help file by typing **? setwd()** into ℝ. You can use that question mark syntax to get help on any ℝ command.

To actually use a package (once it is installed on your computer, using the above instructions), you can call the package using the `library()` command in ℝ.

```
> library(foreign)
```

The package is now ready to use. Now, let's try reading in some data.

3.2 Reading in (and Writing Out) Data

Here's an example dataset that includes six variables describing four patients who entered a hospital for treatment.

```
patient,dob,entry,discharge,fee,sex
001,10/21/1946,12/12/2004,12/14/2004,8000,1
002,05/01/1980,07/08/2004,08/08/2004,12000,2
003,01/01/1960,01/01/2004,01/04/2004,9000,2
004,06/23/1998,11/11/2004,12/25/2004,15123,1
```

We can read this data into ℝ from an outside "patient.csv" file and display the resulting dataframe.

```
> data=read.csv("patient.csv")
> dim(data)
```

```
[1] 4 6
```

```
> data
```

```
  patient        dob      entry  discharge   fee sex
1       1 10/21/1946 12/12/2004 12/14/2004  8000   1
2       2 05/01/1980 07/08/2004 08/08/2004 12000   2
3       3 01/01/1960 01/01/2004 01/04/2004  9000   2
4       4 06/23/1998 11/11/2004 12/25/2004 15123   1
```

Note that this creates a dataframe called **data** that has a row for each patient and a column for each variable.[3] You can now proceed with analysis on this dataframe as you would any other, just as we did with the dataframes we created above.

When we have created a dataframe (and potentially manipulated it in a number of ways) and want to save our new creation, we can use the `write.csv()` command to store the dataframe as a .csv file in our working directory.

3.3 Data Manipulation and Subsetting

Often, we want to work with some smaller part of a dataset rather than the entire dataset — for example, we may only be interested in looking at a subset of observations or looking only at some variables. ℝ has a number of functions for manipulating and subsetting dataframes that can help us with these tasks. Recall that for dataframes, we can use the `$` symbol

[3] You can call your data anything, but you need to store it as something. I always call my dataframes **data** or some variant thereof for simplicity, but you can use anything you want.

to identify specific variables or use the [,] brackets to index specific observations, specific variables, or both. We will also make use of the ! symbol and the - symbol, which serve as "not" commands. We also will use the subset() command to isolate types of observations and variables. We'll use the patient data we just read in to demonstrate the results of using these commands in various combinations.

```
> data$patient

[1] 1 2 3 4

> data[,1]

[1] 1 2 3 4

> subset(data,select=c(patient))

  patient
1       1
2       2
3       3
4       4

> subset(data,select=-c(patient))

         dob      entry  discharge   fee sex
1 10/21/1946 12/12/2004 12/14/2004  8000   1
2 05/01/1980 07/08/2004 08/08/2004 12000   2
3 01/01/1960 01/01/2004 01/04/2004  9000   2
4 06/23/1998 11/11/2004 12/25/2004 15123   1

> data[1,]

  patient        dob      entry  discharge  fee sex
1       1 10/21/1946 12/12/2004 12/14/2004 8000   1

> subset(data,patient==001)

  patient        dob      entry  discharge  fee sex
1       1 10/21/1946 12/12/2004 12/14/2004 8000   1

> subset(data,sex==2)

  patient        dob      entry  discharge   fee sex
2       2 05/01/1980 07/08/2004 08/08/2004 12000   2
3       3 01/01/1960 01/01/2004 01/04/2004  9000   2

> subset(data,!sex==1)

  patient        dob      entry  discharge   fee sex
2       2 05/01/1980 07/08/2004 08/08/2004 12000   2
3       3 01/01/1960 01/01/2004 01/04/2004  9000   2

> subset(data,sex==2,select=c(dob))
```

```
         dob
2 05/01/1980
3 01/01/1960
```

The subset() command is incredibly powerful. Note, however, that none of these subsets were stored, they were simply displayed. If you want to further use a subset of a dataframe, it is necessary to store that subset (of observations and/or variables) as a new object:

```
> data.females=subset(data,sex==2)
```

It is also possible to temporarily use a subset of data using the with() command:

```
> with(data, mean(fee))

[1] 11030.75

> with(subset(data,sex==1), mean(fee))

[1] 11561.5

> with(subset(data,sex==2), mean(fee))

[1] 10500
```

Note that here, you don't need to use the $ operator or [,] indexing brackets because the with() command tells ℝ which dataframe to use when we call the fee variable.

3.3.1 Missing Data

Missing data, represented in ℝ using NA, will inevitably create problems for you at some point during this course. Whereas some statistical packages, like SPSS, SAS, and Stata, will automatically drop missing values from many types of analysis (leaving the original dataframe or matrix intact), ℝ will often choke when you have missing values. This typically causes confusing errors about the length of vectors or dimensions of matrices. It is also good statistical practice to be fully aware of the amount and locations of missing data in your dataset because the standard practice of dropping observations with missing values can often create worse inferential problems (like selection bias, misleading results, etc.) than using a number of other strategies for addressing missing data.

Perhaps most confusing is that mathematical operations involving missing values all produce missing values, even if you might expect them to treat the NA as a 0:

```
> missing=NA
> value1=2
> value2=3
> value1+value2+missing

[1] NA
```

Something similar happens with vectors, matrices, and dataframes. Take, for instance, a vector (v1) and let's remove two values and see the effect on various functions:

```
> v1=c(1,2,3,4,5,6,7,8,9,10)
> length(v1)

[1] 10

> sum(v1)

[1] 55

> mean(v1)

[1] 5.5

> v2=c(1,2,3,4,NA,6,7,8,9,NA)
> length(v2)

[1] 10

> sum(v2)

[1] NA

> mean(v2)

[1] NA
```

You can also identify which values in a vector, matrix, or dataframe are missing using the logical function `is.na()`. You can also nest this command inside other commands. The resulting vector of `TRUE` and `FALSE` logicals can also be treated mathematically, such as using `sum()` to count the number of missing values in the vector.

```
> is.na(v2)

 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE

> sum(is.na(v2))

[1] 2
```

We could simply remove the missing values for the purposes of performing the calculation. For most functions, this can be accomplished by adding a `na.rm=TRUE` argument within the function. Or, we could create a subset of the original data that omits the missing values using the `na.omit()` function.

```
> v3=na.omit(v2)
> length(v3)

[1] 8

> sum(v3)

[1] 40

> mean(v3)

[1] 5
```

Using the `na.omit()` function changes the length of the original vector. So, while `cor(v1,v2)` will give us a resulting correlation between our original two vectors of `NA`, if we attempt to calculate the correlations of `cor(v1,v3)` or `cor(v2,v3)` ℝ will produce an `incompatible dimensions` error because `v3` has a length of 8, while both `v1` and `v2` have length 10. In practice, you may want to consider imputing missing values rather than simply removing missing values to avoid these technical errors and larger inferential problems. That, however, is the subject for another discussion.

4 Randomness

One of the very convenient sets of functionality in ℝ is its ability to generate (pseudo-)random numbers and to conduct sampling. Details about the different random number generates can be found here: `http://stat.ethz.ch/R-manual/R-devel/library/stats/html/Distributions.html`. Random numbers can be generated according to a number of different distributions including (using the attendant command):

- Normal/Gaussian: `rnorm()`
- Chi-squared: `rchisq()`
- Student's t: `rt()`
- Uniform: `runif()`

Each of these (and the other) distributions also have built-in functions for producing density functions, quantiles, and p-values using commands of the form `dnorm()`, `qnorm()`, and `pnorm()`, respectively. True random numbers and sequences can also be obtained using the `random` package, which makes calls to the random number generates at `http://www.random.org` instead of the built-in R pseudo-random number generators. Random integers can be extracted using the `random` package or by rounding the output of one of the `runif()`-type functions.

Besides generating random numbers, ℝ also provides the ability to sample values from a vector or observations from a dataframe using the `sample()` command. Sampling can be conducted with or without replacement and involve samples of any size:

```
> set = c(1,2,3,4,5)
> sample(set,5,replace=FALSE)

[1] 2 4 5 3 1

> sample(set,5,replace=TRUE)

[1] 2 5 3 5 4

> sample(set,10,replace=TRUE)

 [1] 2 5 4 1 4 1 3 1 2 1
```

Sampling commands could be used to conduct simulations, including bootstrapping and permutation procedures, as well as Monte Carlo simulations (the latter, in combination with the random number generation functions).

5 Plots

The real power of ℝ is its sophisticated graphical functionality. ℝ comes with a number of built-in plotting commands; other plots can be produced using a variety of packages or by writing your own code. Let's discuss a couple of the basic ℝ plots; these should be familiar to you from your previous statistics and mathematics training.

- `barplot()`
- `hist()`
- `pie()`
- `plot()`
- `boxplot()`

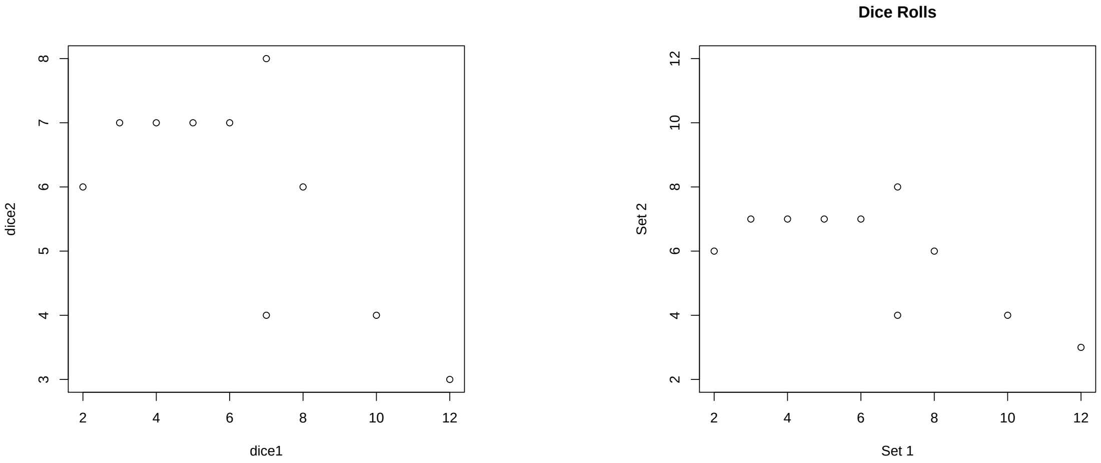
5.1 Plotting Parameters

All plots can be modified using a common set of parameters described in detail at `http://stat.ethz.ch/R-manual/R-patched/library/graphics/html/par.html`. Some of the key parameters are:

- `main`: Graphic title
- `xlab`: X-axis label
- `ylab`: Y-axis label
- `xlim`: X-axis minimum and maximum
- `ylim`: Y-axis minimum and maximum
- `pch`: Bullet style
- `lty`: Line type
- `lwd`: Line thickness
- `col`: Color

Going back to our dice rolling data, we can create a simple scatterplot of the two sets of dice rolls and then modify that plot using the the graphical parameters just described. The parameter arguments can either be comma-separated on the same line, or comma-separated on separate lines for ease of read. Let's compare the basic scatterplot `plot(dice1,dice2)` to a plot with various plotting arguments specified:

```
plot(dice1,dice2,#Data to plot
main="Dice Rolls",#Title
xlab="Set 1",#X-axis label
ylab="Set 2",#Y-axis label
xlim=c(2,12),#X-axis dimensions
ylim=c(2,12),#Y-axis dimensions
pch=21 #Solid points
)
```



This is a fairly simple set of changes, but it is also possible to change the size, color, and appearance of almost any feature of the plots, as well as design a matrix of plots as a single image output using the `par()` and/or `layout()` commands.

One particularly neat feature of ℝ is that plots can be modified and combined to create informative presentations of multiple types of data. After you have created a plot, it is possible to add additional data to the plot using one of a number of commands:

- `lines()`: Add a line to a plot using starting and end points
- `abline()`: Add a horizontal, vertical, or slope-intercept line to a plot
- `points()`: Add point(s) to a plot
- `text()`: Add text to a plot

5.2 Saving Plots

Once you have created a plot, you will likely want to save it outside of ℝ (for example, to include in your problem set). The easiest way is to simply copy and paste the graph from ℝ into some other program. A more flexible way is to sandwich the `plot()` command between two lines of code, like so:

```
pdf("hist1.pdf")
hist(dice)
dev.off()
```

This saves your plot as a PDF file, with the name you specify, in your working directory. You can also use analogous commands — `jpeg()` and `png()` — to save your plot in other image formats.

6 Basic Programming Tools: Logics, etc.

As I've stated above, R is a programming language. Thus, it will be helpful to understand some basic programming to help you move beyond the relatively simple tasks we've learned about so far. I will focus on four areas:

1. Logicals
2. Functions
3. Apply
4. Loops

6.1 Logicals

Logicals allow you to test the equivalence of different ℝ objects in any typical mathematical fashion:

- Equal to: `==` [4]
- Greater than: `>`
- Less than: `<`
- Greater than or equal to: `>=`
- Less than or equal to: `<=`
- Not: `!`

So, we can use these to produce `TRUE` or `FALSE` results from ℝ, which is helpful when building complex programs. Generally it is not terribly helpful to test a simple logical like `2>1`, so instead these logical statements are often used as conditions for performing further operations inside an `if()` statement.

6.1.1 IF-THEN, ELSE

Logicals on their own, are not that helpful. Rather, they are helpful for building complex programs with subroutines that are only run under particular conditions. We could, for instance, set the value of one variable depending on the value of another, or only conduct an operation on an object if it is non-missing. While some programming languages require you to type "then" in order to actually execute the second part of an if-statement, ℝ does not require this. Take a look at the following simple examples:

```
> x=2
> if (x==1) q="FAIL"
> if (x==2) q="SUCCESS"
> q
```

```
[1] "SUCCESS"
```

```
> if (is.na(x)) q="FAIL"
> if (!is.na(x)) q="SUCCESS"
> q
```

```
[1] "SUCCESS"
```

These logicals can also be combined with the AND `&` or OR `|` operators to produce logicals that meet more than one criterion.

[4] Note the double-plus (==), not to be confused with the single-plus (=), which is used to store a value.

```
> x=2
> q=NA
> if (!is.na(x) & x>2) q="SUCCESS"
> q
```

```
[1] NA
```

```
> if (!is.na(x) | x>2) q="SUCCESS"
> q
```

```
[1] "SUCCESS"
```

Here we can see q is not set to equal SUCCESS under the first statement because while x is non-missing, x is not greater than 2. Under the second statement, however, where we use the OR operator q is set to SUCCCESS because x is non-missing, so it does not matter what the value of x is — only one of the two logicals in parentheses needs to be true.

While we can string an infinitely long string of `if()` statements in a row, sometimes we do not need ℝ to evaluate all of the statements — for example, we may want to stop evaluating the `if()` statements once one of them is satisfied. We can use the `else if ()` and `else` commands in these instances.

```
> a=4
> if (a==1) x="Stopped at 1" else
+ if (a==2) x="Stopped at 2" else
+ if (a==3) x="Stopped at 3" else
+ if (a==4) x="Stopped at 4" else
+ if (a==5) x="Stopped at 5" else
+ if (a==6) x="Stopped at 6 or larger"
> x
```

```
[1] "Stopped at 4"
```

You can see here that even though we had two lines of code after the:

```
if (a==4) x="Stopped at 4" else
```

command, those lines were not evaluated because a was set to 4. Thus, we don't have to spend computing time and power to produce a logical TRUE or FALSE for each if-statement, only for the relevant if-statement.

6.2 Functions

While ℝ and its supplemental packages provide an enormous amount of functionality, you sometimes feel compelled to write additional functionality to serve a particular need. In this case, you'll need to use the `function()` command. Let's say, for instance, that we had calculated daily high temperatures for a week in Fahrenheit, but needed them in Celsius. We could perform the conversion manually on each score:

```
> temp1.f=65
> temp1.c=((65-35)*5)/9
> temp1.c
```

```
[1] 16.66667
```

Or we write a function to conduct the conversion without having to retype the math for each score:

```
> temps=c(65,68,72,76,69,78,68)
> ftoc=function(f) {
+         c <- ((f-35)*5)/9
+         print(c)
+ }
> ftoc(temps[1])
```

```
[1] 16.66667
```

6.2.1 Apply

This saves us a bit of time because it simplifies what we have to type, but it's still not very efficient. If we wanted to speed up the process even further, we could use ℝ's `apply()` functions to automatically perform the function on every element in our `temps` vector.

```
> celsius=sapply(temps,FUN=ftoc)
```

```
[1] 16.66667
[1] 18.33333
[1] 20.55556
[1] 22.77778
[1] 18.88889
[1] 23.88889
[1] 18.33333
```

```
> celsius
```

```
[1] 16.66667 18.33333 20.55556 22.77778 18.88889 23.88889 18.33333
```

We can also use `apply()` built-in functions to matrices, vectors, and dataframes. This can be helpful if you want to calculate summary statistics for a number of different subgroups of observations within your dataset. Apply can also be used to do a lot of other things (including some cool and interesting things), but I won't go into the details here.

6.3 Loops

The last important piece of ℝ programming that you should know about is looping. Like `apply()`, loops allow you to perform an otherwise tedious task a number of times on the same or different data, similar to using `apply()`. Loops and `apply()` are what really sets ℝ apart from its competitors like SPSS, SAS, and Stata. There are several different types of loops that can be used, which are describe in detail at `http://cran.r-project.org/`

doc/manuals/R-lang.html#Looping. The simplest loop can be used to perform the same temperature conversion we used above.

```
> celsius=NA
> for (i in 1:length(temps)) {
+         celsius[i]=((temps[i]-35)*5)/9
+ }
> celsius
```

```
[1] 16.66667 18.33333 20.55556 22.77778 18.88889 23.88889 18.33333
```

Here, we tell ℝ that we want to conduct a task for *i* number of times, specified by 1 to the length of our temps vector. For each iteration of the loop, we take the i^{th} element of the temps vector, perform the temperature conversion calculation and store the resulting value in the i^{th} element of the celsius vector. When the loop completes, ℝ displays nothing, so we have to call the celsius vector. If we didn't need to store each converted value, we could also use the print() command to simply display the resulting calculations like so:

```
> for (i in 1:length(temps)) {
+         print(((temps[i]-35)*5)/9)
+ }
```

```
[1] 16.66667
[1] 18.33333
[1] 20.55556
[1] 22.77778
[1] 18.88889
[1] 23.88889
[1] 18.33333
```

The print() function will be helpful if you run very large or complicated loops.

But now we've seen multiple ways to apply functions to objects. Which should we use in which situation? Given that we have nearly unlimited computing power and time, the distinctions between these methods are fairly small. apply() functions tend to be faster than loops, but that only matters if the task you're performing is very complicated. Loops are more versatile because you can perform multiple tasks within a single iteration of a loop, as well as nest loops inside one another. For example, if you needed to perform a task on each element of an *i*-by-*j* matrix, you could nest a loop indexed by *j* within a loop indexed by *i*, or vice versa. The final choice of how to program depends on the needs of the particular task you're aiming to complete. Often there many, many ways to accomplish the same task, which makes ℝ uniquely versatile among statistical packages.

7 Conclusion

So now you've learned the basic information you need to know to use ℝ for this class. Hopefully this tutorial has helped you be less intimidated by and frustrated with the language and this course as a whole. Best of luck and let me know if you have questions!