Lecture 1: Getting Started in R

The first lecture is intended to provide you the basics for running R in Windows/Mac

Outline:

1. An Introductory R Session
2. R as a Calculator
3. Getting Help and Loading Packages

Objectives:

By the end of this session students will be able to:

1. Identify the components of the R interface for Windows
2. Conduct standard arithmetic calculation: both numerical and matrix
3. Access R help and load packages in R

Why should we learn R?

1. It's free!
2. Take advantage of the availability of new and cutting edge methods in the field
3. Understand the literature well since more and more people report their results in the context of R
4. Online support and resources readily available
5. *Highly* extendable:

   Interactive Web Search (web scraping)

   Use in Unix environments for scientific computing: BU Shared Computing Cluster (SCC)

   Can be used in conjunction with Python, SAS, Excel, LaTeX, JAGS, OpenBUGS, others

1.1 Downloading, Installing and Running R

An up-to-date version of R may be downloaded from the site called CRAN (this stands for Comprehensive R Archive Network): http://cran.r-project.org/. Installation instructions are also provided on the website. In the following instructions, it is assumed that you are using a Windows system. You can click on "Download R for Mac" or "Download for Windows" and follow the instructions to download R.

Once you have installed R, you will see a blue desktop icon. To run R, just click on this icon. Alternatively, you can click on the R icon under Program by going to Start if you don't see it on your desktop. Then you will see a blank line with a **command line prompt** (>) symbol in the left hand margin under the statement. This prompt invites you to type in your commands or expressions. For example,

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

To quit the R program, the command is

```
> q()
```

(Also take note of the Stop Sign button. This will come in handy sometimes.)

And we usually don't save the *workspace* (all commands we've issued and objects saved since we started the program,), but instead save our code/script in a file with a **.R** suffix.

If we want to execute our command from a document script, not the console, we can type **command(⌘)-return** (on a Mac); on a PC, highlight the code and right-click mouse to execute, or type **Ctrl-**R while the cursor is on the line you want to execute.

Type the up arrow or down arrow ↑↓ to toggle between previous commands.

To enter a line of comments, type the hash tag (pound) symbol: #


STUDIO

There is a more flexible version of R that some people prefer called *RStudio*. It is a little more GUI (graphical user interface) than the traditional R package. It looks nice, and you can make very nice documents with it:

- Markdown with LaTeX code embedded
- pdf documents
- html webpages

RStudio is also able to interface with *Github*, for those interested in software development and version control. The package is quite handy, and mostly compatible with standard R, however there are a few commands that work for RStudio that do not work for standard R.

Here's a decent online tutorial for using the program:
https://www.youtube.com/watch?v=uwlwNRbaKMI

R Markdown
RStudio supports creation of PDF and HTML web documents for reports that have R code and graphics embedded in the document (very nice for reproducible reports and tutorials)

Shortcuts
R has many shortcuts you can use that will really make your life easier.

For example, you can clear the console of all previously executed commands if it gets cluttered up.

On a Mac, type **command(⌘)-option-l**.
That last symbol is the letter L (ell), not I (eye).
On a PC, type **control-l**.

This site has a full list of command shortcuts for RStudio:
https://support.rstudio.com/hc/en-us/articles/200711853-Keyboard-Shortcuts
Most of these are also used in standard R.
In RStudio, go to top ribbon, then ***Tools → Keyboard Shortcuts Help***

1.2 R as a calculator
The simplest way to use R is treating it as a calculator. For example, if we want to know what two times two is, you can type

```
> 2*2
[1] 4
```

or

```
> 2*2;
[1] 4
```

The notation [1] indicates that the result is the first element (from a possibly larger list). This is useful later when we have many elements for one variable such as a **vector**. If you carefully compare this to the previous command, you will notice that we added a semi-colon (;) here. However, the result remains the same since the semicolon is used as the separator for multiple commands on the same line.

Sometimes, you may want to use **built-in functions** in your calculations.

```
> log(10)
[1] 2.302585
> exp(2)
[1] 7.389056
> sqrt(4)
```

```
[1] 2
> abs(-4)
[1] 4
>
```

Exercise: Type in the following commands and describe your observations.

```
> # case 1 : 1+2
> 2+2; 2*3; 2/5
> # case 2
> 8/2-2*(2-3)
> # case 3:
> 3*5  * 4   /2
```

(as always, remember your order of operation: "PEMDAS.")

A note on machine precision

R, like any computer program, necessarily truncates values after some number of decimal places, and sometimes represents numbers as 0 when it really means 0.000000000001.

For example **1 + 1.110223e-16** is obviously not 1. But go ahead and type it into R and you get:

```
> 1+1.110223e-16
[1] 1
```

Usually this sort of thing isn't a problem, but it's something you need to keep in mind in case you are performing computations with either very large or very small numbers.

Assignment Function

In the calculation, we may want to save the intermediate results for later use. This can be achieved by assigning values to symbolic variables using an "assign" function. So to create a *scalar* (non-vector) constant **x** with value of 2, we type

```
> assign("x", 2)
```

This can also be simplified by using the operator <-

For example,

```
> x <- 2
> x
[1] 2
```

We can also assign multiple objects the same value:

```
> x <- y <- 2
> x
[1] 2
> y
[1] 2
```

Assignment is the bedrock of R, and is indispensible for almost everything we end up doing in this class. For example, if we want to analyze and export as a spreadsheet only a section of rows in a dataset, we might assign rows 1 through 10 of a dataset to a new "object" we can name, for example ***data.subset***.

Note that the symbol "<−" is made up from "less than" and "minus" with **NO space** between them. The assignment operator can be used in the other direction, ->, but this use is not really standard.

Some other commonly used operators are:

1. arithmetic	+ - * / ^	
2. relational	> >= < <=	
	==	(equals)
	!=	(not equal)
3. logical	!	(NOT)
	&	(AND)
	|	(OR)
4. assignment	<- ->	
5. create a sequence	:	

A complete listing can be found here:
http://stuff.mit.edu/afs/sipb/project/r-project/arch/i386_rhel3/lib/R/library/graphics/html/plotmath.html

Once you assign an object a designation, it stays in the working memory until you close the program. To see what objects are in the working memory, type `ls()`, or select `Show Workspace` command from the dropdown menu.

You can clear the workspace by the dropdown menu, or by typing:

```
> rm(list=ls())
```

Let's look at some examples to see how these operators work.

```
> 1>=3
[1] FALSE
> !(1>3)
[1] TRUE
> (3 != 1) & (2 >= 1.9)
[1] TRUE
```

R allows us to create logical vectors and to manipulate logical quantities as well. To create logical vectors, you may use TRUE, FALSE, or NA (for **missing** / ***n*ot *a*vailable**) directly, or type in the condition/logic operation. Note that in order to be used in arithmetic calculations, R treats TRUE as 1 and FALSE as 0.

```
> y <- c(TRUE, FALSE, 5 > 2)
> y
[1]  TRUE FALSE  TRUE
> sum(y)
[1] 2
```

Exercise: If we type the following commands into R. What will (x, y, z, w) be?

```
> x <- !(5>=3)
> y <- ((2^4) > (2*3))
> z<- x|y
> w <- x&y
```

Vectors

Vectors are variables with one or more values of the same type such as numerical, logical or character. To create a vector named **x**, say, consisting of four numbers, 1.2, 2.3, 0.2 and 1.1, we can use the R command

```
> x <- c(1.2, 2.3, 0.2, 1.1)
> x
[1] 1.2 2.3 0.2 1.1
> length(x)
[1] 4
```

Concatenation

Here, we introduce function `c( )` where c stands for *concatenation* (joining numeric, character or logical values together). Note that it has to be round parentheses, i.e. (and), not square ([,]) or curly ({, }) brackets. Also, we can use the function `length()` to find out how many elements the vector has. If we want to select only some elements in the vector, then we can use indices. For example, if we want to know what the last three elements are in variable x, then we can type

```
> x[c(2,3,4)]
[1] 2.3 0.2 1.1
```

(Note that to *subset* vectors or datasets we need to use square brackets, but to actually *define* a vector, we use round parentheses—this takes a little getting used to.)

Exercise: What do you see in the screen if we type the following commands?

```
> x[-1]
>
> x[2:4]
```

Based on your observation, what does the negative subscript/index mean? Also, what does the function colon (:) mean? (This one comes in handy when we you learn about loops)

Note that if you use a negative number in your index, don't include positive numbers as well. R won't know what to do. The command:

```
> x[c(-1,4)]
```

will produce an error message.

Vector arithmetic

In some circumstances, we may want to apply certain operations or calculations element-by-element in the vector. For example, we would like to create a vector, say **y**, that has elements that are each 2 times the corresponding element of **x,** and plus 3. Then instead of going element-wise on every component of x using the command c(2*x[1]+3, 2*x[2]+3, 2*x[3]+3, 2*x[4]+3), we can just type

```
> 2*x+3
[1] 5.4 7.6 3.4 5.2
```

The logical vector may also be used to select subsets of a dataset. For example, in the previous exercise, we saw that `x[c(2,3,4)]`, `x[-1]` and `x[2:4]` work exactly the same. You can also use

```
> x[c(FALSE, TRUE, TRUE, FALSE)]
[1] 2.3 0.2
```

Another example: the command

```
> x[x>1]
[1] 1.2 2.3
```

only lists the elements in x greater than 1.

In other words, R can evaluate functions over entire vectors. The elementary arithmetic operators will function as per their usual meaning (+ - / etc.).

In addition to these arithmetic operations, some vector functions are also available. For example,

```
max(x)
min(x)
sum(x)
mean(x)
median(x)
range(x)
var(x)
```

Matrices

Matrices are a two-dimensional generalization of vectors. There are many ways to create a matrix. They contain only numeric values. For example:

```
> x <- matrix(c(1,2,3,4,5,6), ncol=2)
>
>
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

where the parameter **ncol** is the number of columns in this matrix, and the numbers are entered by default column-wise. There are two indices, row and column. We can use the commands below to extract cells of the matrix:

```
> x[3,2]
[1] 6
> x[3,]
[1] 3 6
```

These commands can be used to call the element of 3rd row and 2nd column in matrix x. If we don't specify any element in the index, R will return all the elements.

```
> p<- matrix(c(1,2,3,4,5,6), nrow=2)
> p
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Other commonly used approaches to create matrix are `cbind()` and `rbind()`, which are generalizations of the `c()` function described earlier. The `cbind()` function stacks vectors as columns, the `rbind()` function stacks vectors as rows.

For example, the following commands will create the same matrix as **x**.

```
> cbind(c(1,2,3), c(4:6))
> rbind(c(1,4), c(2,5), c(3,6))
```

Exercise: Suppose we have two vectors, x and y, defined as follows.

```
> x <- c(-3:3)
> y <- c(2, 5, -6, 3, -2, 10, -4)
```

(1) Create a matrix, call it *z*, that is composed of *x* as the first column and *y* as the second column.
(2) What's the mean of the first row of matrix *z*? How can we program this?

Aside: Types of DATA

R has data "types," which are sometimes manipulated differently from each other under certain operations. Most data types are: *numeric* (or integer), *factor* (non-numeric data, for example, the cancer grade of a cancer patient), and *character* (their name, or address).

Data Frames are like matrices, but instead of numbers, the entries can be numbers or factor variables. There is also types *list* and *array*, but we will not cover those here.

A nice R command is the `str()` command (which stands for **str**ucture). This takes an object and tells you what data type it is:

```
> k <-3
> k
[1] 3
>
> str(k)
  num 3
>
> w <-"Homer"
> w
[1] "Homer"
>
> str(w)
   chr "Homer"
```

(The command `class()` does the same operation, but outputs the entire name of the data type.)

Matrices vs. Data Frames:

Matrices and data frames are two ways to structure 2-dimensional information. They are different in a few ways. Generally, use data frames if the variable types are not all **numeric**.

```
> matrix.1<- matrix(1:16,4,4)
>
> matrix.1
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
>
> str(matrix.1)
 int [1:4, 1:4] 1 2 3 4 5 6 7 8 9 10 ...
>
> is.matrix(matrix.1)
[1] TRUE
>
> is.data.frame(matrix.1)
[1] FALSE
```

```
> data.1 <- as.data.frame(matrix.1)
> data.1
  V1 V2 V3 V4
1  1  5  9 13
2  2  6 10 14
3  3  7 11 15
4  4  8 12 16
>
> str(data.1)
'data.frame':  4 obs. of  4 variables:
 $ V1: int  1 2 3 4
 $ V2: int  5 6 7 8
 $ V3: int  9 10 11 12
 $ V4: int  13 14 15 16
>
> object.size(matrix.1)
264 bytes
> object.size(data.1)
1048 bytes
```

Datasets in R

Around 100 datasets are supplied with R, and others are available.

To see the list of datasets currently available use command

```
data()
```

We will first look at a data set on CO2 uptake in grass plants available in R.

```
> CO2
```

(note: capitalization matters here; also: it's O (the letter), not 0 (zero))

To get more information on the variables in the dataset, type in

```
> help(CO2)
```

If you know the name of the column you wish to evaluate, you can type the name of the dataset followed by a dollar sign ($) and then the name of the variable.

Exercise : Evaluate and report the mean and max and min of the variables "`Concentration`" and "`Uptake`".

Getting Help

Before we get deeper into the use of R, it is good to know how to seek help when we get stuck. Two functions are illustrated here. If you know the name of a function or the topic, you may use the function `help(…)` with your function name or topic inside the parenthesis. For example, if you are interested in the function `plot()`, you can type

```
> help(plot)
```

and it will pop-up a help manual for plot function. This can be done more quickly by typing a question mark in front of the function in question.

```
> ?plot
```

Sometimes, you may want to find something related to a certain keyword. Then you may find `help.search()` useful. Function `help.search()` will search for all the functions that have the word you specified in their help document such as name, title, concept, keyword. For example,

```
> help.search("sort")
```

will list all functions that have the word 'sort' as an alias or in their title. Function `help.search()` also has a shortcut through two question marks (??). If you feel like knowing more about the help function, you may type `help(help)`. Other than the official help pages, you can explore the Internet. There are really a ton of resources.

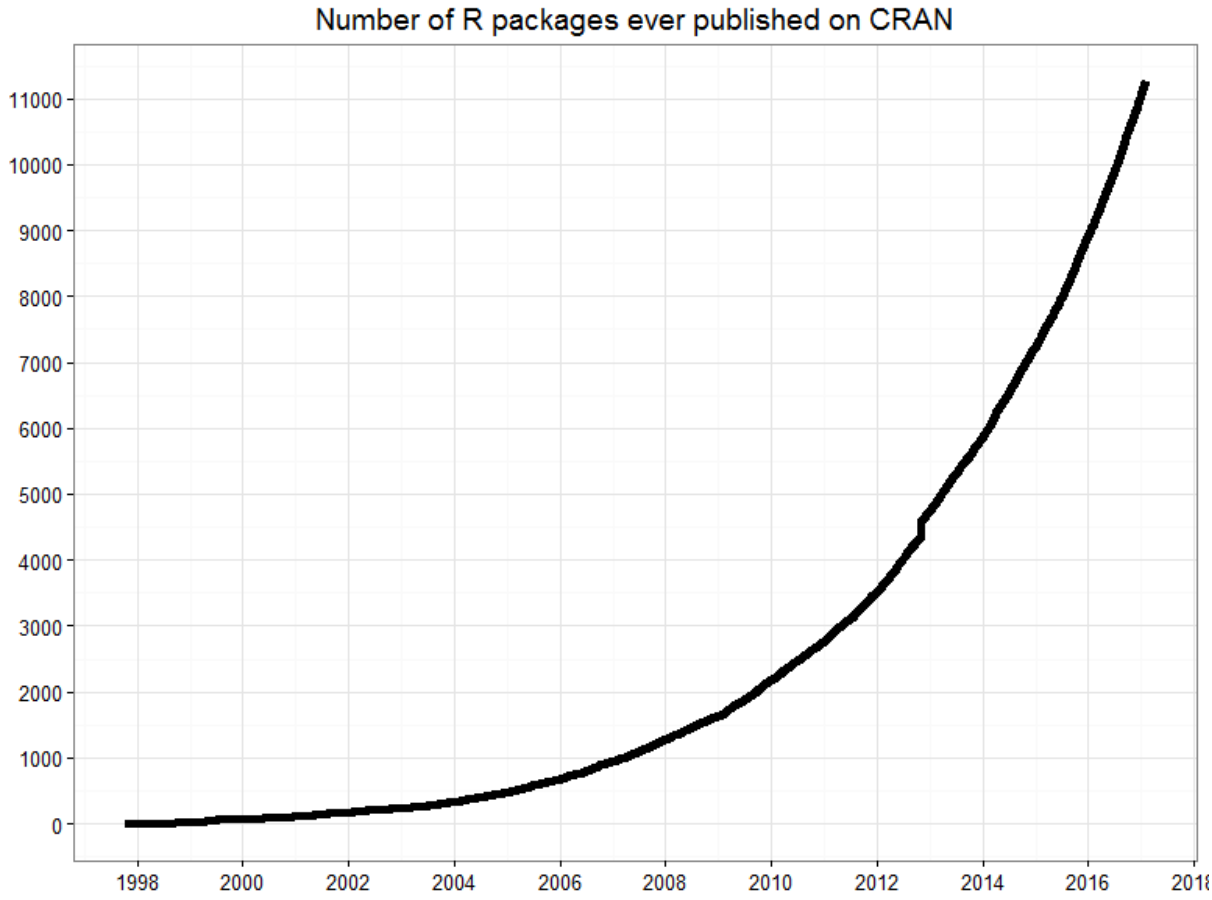
You can search any command or function you want to know more about, and there are numerous online forums with detailed tutorials and examples: https://www.r-bloggers.com/

If you want help on an operator, such as the plus symbol, you need to use `help()` and not just the `?`<command>. You also need to use quotes. For example:

```
> help("+")
```

Packages

R is open source software, so many so-called *packages* are freely available. Packages are user-created R functions, sometimes containing datasets and long-form examples of the function's use in .pdf documents. *However*, not every package is installed or loaded when you download and open R.



If we would like to use a certain package that is not yet installed in your base R suite, then you need to install it and then load it into R for use.

Packages that have been vetted and shown to work properly and obey all the requisite R community guidelines go the **CRAN** network, a series of servers hosting all R programs and packages.

For example, you may want to use package **ISwR**, here is the procedure:

1. Install package ISwR:
   Click **Set CRAN Mirror…** in the **Packages** options
   -> Pick the closest server site, eg **USA (MA),** and click **OK**
   -> Click **Install Package(s)…** in the **Packages** options
   -> Pick ISwR

Once it is downloaded you will see something like the following statement. Note that I have deleted some output to save space. Don't panic if you see a warning here; you may ignore this for the time being.

```
   ….
package 'ISwR' successfully unpacked and MD5 sums checked
The downloaded packages are in
C:\Users\ctliu\AppData\Local\Temp\RtmpjNPIgz\downloaded_packages
```

2. Load the package ISwR

Once you install the package, you still need to load it into R before use. To load the package, type

```
> library(ISwR)
```

Now package ISwR is ready for you to use.

You can also install directly from your command line by typing

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

As of last year there are over **10,000** R packages (really).

This post describes ways to find whatever package you are looking for:

https://www.r-bloggers.com/cran-now-has-10000-r-packages-heres-how-to-find-the-ones-you-need/amp/

Saving

You can save your entire work history, along with any functions you may have written and code you executed, into a file for later use. Use the command

```
> save.image(file="yourpath/file_name.RData")
```

and you will have a file called *file_name*.RData in your computer. The icon will look like this:

or this:

You can load this file by typing

```
> load(file="yourpath/file_name.RData")
```

and then all the previously saved objects/functions/data will be present in your R working directory.

Recap:

- Why R?
- Using R as a calculator
- Variable assignment
- Vector & matrix arithmetic
- The 'help' function
- Installing packages