Bioinfo-R-matics

This hands-on workshop will provide training on the most common aspects of R. Participants should have an understanding of general biological concepts and statistics. This course in NOT about teaching you statistics, but simply about teaching you how to effectively interface and work with R. By combining lectures and hands-on activities participants will leave with an ability to take data tables from excel and produce publication ready figures and analysis. We will also cover some the libraries and tools available in Bioconductor, which is a repository for bioinformatics analysis tools written in R.

The information in the document comes from activities used by R. Kodner and B. Miner at WWU in our Biometrics course. The main source for this workbook comes from:

- **Advanced R by Hadley Wickham**, freely available at http://adv-r.had.co.nz/
- **Getting Started with R**, Beckerman, and Petchey http://www.r4all.org

What is R

R is an open-source, free program for statistics that is maintained by a small group of statisticians. The program comes with a core set of "packages" that enable certain functions. You can add additional functions by installing and loading other "packages". You can perform statistical tests, create complex mathematical models, and graph in R. For graphing, there are basic graphing functions that will build graphs for you. However, you can also build graphs from scratch and modify every aspect of a graph. See the R website for much more information. R can be downloaded here:

R studio, a very convenient interface for R can be downloaded here: http://www.r-project.org/. http://www.rstudio.com/products/rstudio/. You can work with either the standard R console or R studio in this tutorial.

Introduction to R – The Basics

In these exercises **>** represents a new console line.

```
> ls()
```

Code with out a **>** is meant to be added to a script.

```
head(my.data)
```

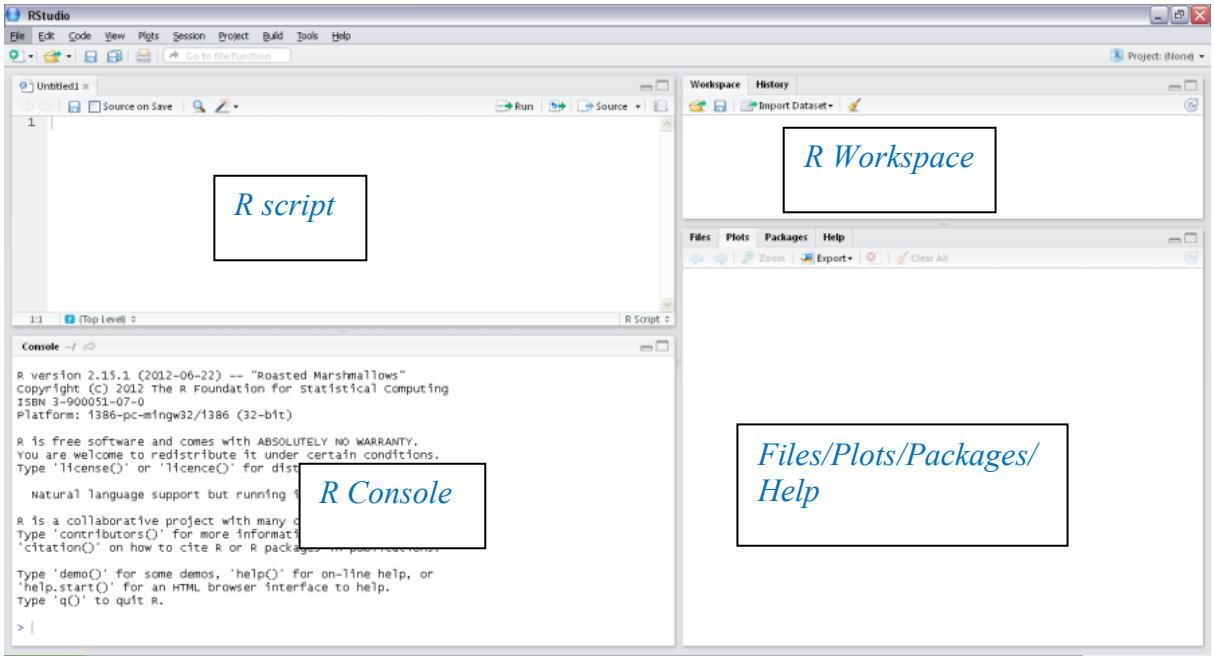
The **#** is used to add comments that R will not execute, so you don't have to type them or what follows them on a line.

Downloading the R software

R is an open-source program available from the site below. It is regularly updating as user improve it. Individual users create specialized libraries and packages that can be downloaded and added as necessary to your version of R

Download R from http://cran.us.r-project.org/

The R interface is not particularly easy to use. Several R "skins" have been developed. One of the more useful is R Studio. This package allows you to save code you have written, work with figures and install packages among other things.



R Script: This is the area where you write commands, programs, etc. You can run single lines of code or code blocks using keyboard shortcuts or buttons in the header. The entire script can be saved as a *filename*.R file, allowing you to use it again later.

Console: This is where the commands from your Script are actually executed. Nothing here is saved. You can type individual commands here, but you can't recapture what you have done as a script. Commands are executed when you choose ENTER. The up arrow on the keyboard can be used to select and repeat earlier commands in the console. Similar to how the Terminal operates.

Workspace: This is where objects created by your script are temporarily stored. These objects are lost when you exit the program. It can be useful to view you objects and review variable names and types in this window.

Files/Plots/Packages/Help: *Plots* are where graphics are displayed. You can use the arrow to cycle between your graphs. None of these are saved unless you export them. You can export images as PDFs from here or add code to your scipt to save images as svg, png, etc. *Packages* shows packages you have download and installed to your base R program. If you need to add a package, you can Google the one you want, download it, and choose "Install packages" from the *Packages* menu. The packages will be saved on your computer, but will have to be called in your code each time you need them. *Help* provides access to information about commands.

Basic R Functionality

To begin let's just use R like a calculator to get an idea of how the program works. Type in the following on the console and observe the output. Reminder: you will not type the leading **>**

```
> 2 + 2
> 6 - 4
> 3 * 6
> 4 / 8
> 2^3 #exponent
> 2 - 3 * 4
> (2 - 3) * 4 #parentheses change the order of operations
> sqrt(2) #square root
> pi
> sin(pi)
> exp(1) #e^1
> log(10) #natural log
> log(10,10) #log base 10
```

Errors and warnings

R will provide errors or warnings when you do something that it doesn't like. Typically the error or warning messages are helpful. Thus, read them carefully and use this information to help correct the problem. If you don't understand what the error means you can often find help by copying the error or warning message and pasting it in Google. Let's create a couple of errors and warnings so you can see what they look like–by the end of the class you will have seen many of these.

```
> squareroot(2)
> sqrt 2
> sqrt(-2)
```

If you forget to add the closing parentheses and hit return then R will just assume that you are going to continue to type more on the next line. You can use ESC to exit and get to a new line.

Try the following.

```
> sin(3
#hit ENTER
> )
#hit ENTER
> sin(3
#hit ENTER
#hit ESC, ESC will also abort a calculation.
```

Assigning names to Objects

Probably the most important thing to learn in R is how to create, name, and assign values to variables. This is what we call creating an R object. Lets start by creating variable and assigning a name and value to it. There are three ways to do this. **Beware that R is case sensitive**.

```
> a=1
> b <- 1
> 1-> c
```

If you type in a, b, or c, which as now objects, R will return the values assigned to each of the objects. In this case, they are all assigned a value of 1. Notice that all three ways produce the same result. The **<-** or **->** syntax is often used because you might also include **=** elsewhere in

your code and get confused. The standard way you will assign names in R is with the object on the left and whatever goes in that object on the right like this:

```
> b <- 1.
```

Now, you can just use the variable or object name, in this case a single letter, to perform calculations. For example:

```
> a + b
```

You can provide any variable with a name and a value. However, it is worthwhile to use names that are meaningful but SHORT, so you can remember what the variable represents but not have to type a lot. For naming variables in R, periods are often used. For example, if I wanted to create a variable with the value of a population mean, I might name it **`pop.mean`**. The value or a variable can take of the form of a number, a word, a vector, a list of numbers or words, a matrix, or data frame for example. You can also create functions and assign them an object name to do simple or complex calculations.

Functions

A function is a set of commands that do something, typically return an answer to the user. A function has two important parts to it, the name and the arguments. The name of a function is what to type and tell R which function you want to use. The arguments, which there might be zero to many, are options (or variables) that are used in a function. All functions will have default setting and you will/can change the defaults using the arguments. All functions have a help page that will tell you how to use them (once you can decipher the way the options are described, this can take a while). You just type a **`?`** in front of the function name, as seen below. This is the first place to go to figure out how to use a function. In this workbook, all functions are shown with **`()`** at the end. In R, a function will always perform it's task on the object or numbers or characters inside the **`()`**.

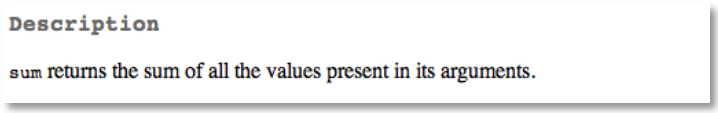
For example, let's use the **`sum()`** function as an example. The name of the function is sum and the argument is placed inside the parentheses.

Type in **`?sum`** in the "R Console" and hit return. A window should pop up. This window provides the user with information about this function.


sum {base} R Documentation

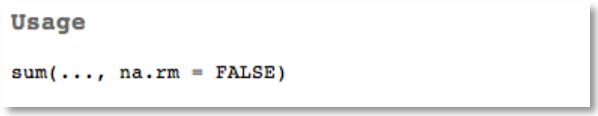
Sum of Vector Elements

First, there is a description of what the function does.


Description

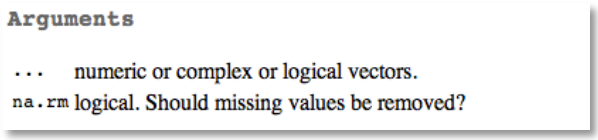
sum returns the sum of all the values present in its arguments.

Next the usage, or how you should type the function,


Usage

```
sum(..., na.rm = FALSE)
```

followed by the arguments for this function.


Arguments

...	numeric or complex or logical vectors.
na.rm	logical. Should missing values be removed?

You can see that there are two arguments. For the first argument, you must supply a vector. For the second argument, you tell R what to do with missing values in the vector. Notice that the second argument shown in the **`Usage, na.rm = FALSE`** has a equals sign. This means that the default for this argument is **`FALSE`**, meaning the program doesn't remove missing values. Because there is a default value we only need to provide the first argument, unless of course you would like to remove missing values that are noted with NA. To try this function but we need some data in a vector.

Data structures in R

Data is stored in R in a number of different kinds of structures that increase in complexity.

	Homogeneous Data	Heterogeneous Data
1d	**Vector**	**List**
2d	**Matrix**	**Data frame**
nd	**Array**	_____

Vectors

A vector is a 1 dimensional group of values that are all the same type: for example, a group of integers or a group of words (categories). Vectors cannot be a combination of different types of values: for example, a group of words and numbers. R will either give you an error, or convert the numbers to characters. Vectors come in two flavors: atomic vectors and lists. They differ in the types of their elements: all elements of an atomic vector must be the same type, whereas the elements of a list can have different types.

The most simple way to create an atomic vector is to just assign a bunch of numbers to an object. Type 1:10, which creates a vector of integers 1 through 10.

```
> 1:10
```

`c()` This function **c**oncatenates numbers or characters separated by commas. This will be super useful when you want to create a vector of characters or words that will be used for figure legends or for choosing colors for a graph, for example. The c() function is very versatile.
`sequence()` This function generates a vector of integers.
`seq()` This function creates a **seq**uence of numbers. Use the ? help to see how to run these functions.

Let's try each of these.

```
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> 0:3
[1] 0 1 2 3
> 8:13
[1] 8 9 10 11 12 13
> 9:2
[1] 9 8 7 6 5 4 3 2
> 0.7:5
[1] 0.7 1.7 2.7 3.7 4.7
```

Now, use the **`sequence()`** and **`seq()`** functions.

```
> sequence(10) #Same as 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> seq(0,3) #Same as 0:3
[1] 0 1 2 3
> seq(5,8,.5)
[1] 5.0 5.5 6.0 6.5 7.0 7.5 8.0
> seq(9,3,-0.8)
[1] 9.0 8.2 7.4 6.6 5.8 5.0 4.2 3.4
> nums<-2:6
> seq(7,12,length.out=3)
[1] 7.0 9.5 12.0
> seq(1,100,along.with=nums) #Vector is set to length of nums
[1] 1.00 25.75 50.50 75.25 100.00
```

Third, let's use the **`c()`** function. This function can create vectors of numbers and characters.

```
> c(1,4,5.4,6)
[1] 1.0 4.0 5.4 6.0
> c("blue","brown","green","grey")
[1] "blue" "brown" "green" "grey"
```

You can also use **`rep()`** – replicator, to make longer vectors of values

```
> rep(c("big", "small"), times=3)
```

Task

1) List 3 ways to create a vector of integers from 15 to 25.

2) Use **`sum()`** to sum the numbers in your vector.

3) How might you find the mean of your vector?

Matrix

A matrix is a 2 dimensional data structure. Matrices have ordered structure, in that data is found in rows and columns, where the row and column are both attributes of a value. Another way to say this is the column and row are important.

To make a matrix, we use the **`matrix()`** function.

Try it

```
> mos <- matrix(c(4,5,12,7))
> mos
> mos <- matrix(c(4,5,12,7), nrow=2, ncol=2)
> mos
> mos <- matrix(c(4,5,12,7), nrow=2, ncol=2,
dimnames=c("a","b"), c("c","d"))
```

What happens? You'll get an error

Here's the fix:

```
> mos <- matrix(c(4,5,12,7), nrow=2, ncol=2, byrow=TRUE,
dimnames=list(c("a","b"), c("c","d")))
> mos
```

What's different? Note the order in which the column and row names are entered

```
dimnames=list(c("row1","row2"), c("column1","column2")))
```

Adding a dimension attribute with the function **`dim()`** to an atomic vector allows it to behave like a matrix.

```
> mo <- c(4,5,12,7)
> mo  #look at your mo object. How many dimensions does it have?
> dim(mo) <- c(2,2)
> mo  #look at your mo object. How many dimensions does it have?
```

If you already have a dataset loaded into R, and it wasn't loaded as a matrix, you can tell R it is a matrix using the **`as.matrix()`** function. We will come back to this function in the future because it will be important for some stats and visualizations.

Vectors are not the only 1-dimensional data structure. You can have matrices with a single row or single column, or arrays with a single dimension. They may print similarly, but will behave differently. The differences aren't too important, but it's useful to know they exist in case you get strange output from a. You can use the structure **`str()`** function to reveal the structure of an R object so you can see what you have.

Try:

```
> str(mo)
```

Task

1) Make a new vector that contains any 6 numeric elements and assign it to an object.
2) Verify the structure of that new object.
3) Make that object a matrix with 2 rows and 3 columns.
4) Verify the new structure.

Array

An array is like a Matrix but is multidimensional. In other words, the rows and columns are meaningful, and there are additional dimension to the data. You create an array with the **`array()`** function. We likely won't use arrays in this class but you can be aware of them.

Data Frame

Data frames are the most common way we will work with data in R and the structure that is most similar to the types of spreadsheets of data you are most familiar with. Essentially, a data frame is a list of equal length vectors. It is 2 dimensional and shares properties of lists and matrices.

You can create a data frame with the **`data.frame()`** function (see a pattern here ☺).
This will take named vectors as inputs.

```
> x <- c("left","center","right")
> y <- c(45,17,53)
> xy <- data.frame(x,y)
```

Now send the data frame to an object and check the structure.

You can also combine data frames using functions call **`cbind()`** and **`rbind()`** **`cbind()`** will combine columns and **`rbind()`** will combine rows.

To demonstrate this, create another data frame call **`ab`** where **`a`** is a vector with there nominal values and **`b`** is a vector with 3 numeric values. Then use **`cbind()`** to create the new combined data frame like this:

```
> abxy <- cbind(xy,ab)
```

Check the structure with:

```
>str(abxy)
```

Task

- Now try making a new combined data frame.
- Data frame 1 should have 3 rows and 2 columns and one vector with nominal values and one vector with numeric values.
- Data frame 2 should also have 3 rows and 2 columns. Both vectors can be numeric.
- Use at least one function besides `c()` to create both vectors.
- Combine data frame 1 and 2 into data frame 3 that has 6 rows and 2 columns.
- Do you run into any errors? What is happening?

Loading Existing Data into R

There are many ways to input data into R. Below we go over some of the most common.

By hand
You just did a number of examples of this.

Import Data
The most common way people import data into R is with the functions **`read.table()`**, **`read.delim()`** or **`read.csv()`**.

We will use this function and import data that is entered into Excel. Now it is time to burn a few rules of the road for transferring files into R into your brains:

NO SPACES

NO SPECIAL CHARACTERS

NO NUMBERS IN COLUMN NAMES

NO WORDS IN COLUMNS WITH NUMERIC DATA

CHECK YOUR SPELLING

KNOW WHERE YOU SAVED THE FILE

To this point we have been simply typing things directly into the R console. From this point forward we are going to begin writing scripts in the "R Scripts" portion of RStudio. This is both excellent practice and most likely the way in which you will work with R on a day-to-day basis. Remember typing individual commands onto the console is great for debugging and testing but there is no official log of your workflow. Creating a script generates a record of your work and makes it reproducible for you and anyone else.

Your First Script

The header: It's important to maintain good documentation. This is an example of a header that you should use on every script you write. This helps you keep a log of your information and makes sharing your code much easier. Type this into the R Script section:

```
# Your Name
# The Date
# Explanation of the scripts function
# (can be more than one line)
```

```
# ------------------------------------------------------------
# Ian Misner
# August 12, 2015
# This is my first R script
# It will read in some files and display basic information
# about the data that was imported
#-------------------------------------------------------------
```

There are a few things we should do to get R ready to execute our script. The first thing that is good to do is to clear R's memory. Add the following to your script:

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

Read this from inside out:

`ls()` = this repeats all the information in R's brain go ahead and type this into the console. See all that stuff?

`list=ls()` = asks R to take the output from **`ls()`** and place it into a list.

`rm()` = asks R to remove all of the items in the list.

The next thing we need to do for simplicity is to move to where our data is located. This is done with the **`setwd()`** command. Which stands for set working directory.

Type this into the console:

```
getwd()
```

Where is R currently looking/located? Write down this output.

This is called the PATH and it's the specific location of files on your computer. Every single file or folder you have has a specific PATH (location) and it always begins with a **`/`** By using the function **`setwd()`** we can change the PATH of R that is using to look for files.

Add this line to your R script:

```
# This function will change R to the folder where the data you
# want to analyze is located. Also the everything inside the < >
# should be replaced with the specifics for your computer.

setwd("<PATH to your R_files folder>")
```

On my laptop the command would look like this:

```
setwd("/Users/UMD-Bioinformatics/Desktop/R_files")
```

Then I can ckeck to make sure R is looking the proper spot using:

```
getwd()
```

After all that this is what my Rscript looks like (for my personal Laptop):

```
# ------------------------------------------------------------
# Ian Misner
# August 12, 2015
# This is my first R script
# It will read in some files and display basic information
# about the data that was imported
#-------------------------------------------------------------


# first I want to clear R's memory of objects
rm(list=ls())

# this tells R where to look for my files
setwd("/Users/UMD-Bioinformatics/Desktop/R_files")

# now check that R is looking in the proper location.
getwd()
```

Now you can save this at myFirstRscript.R and hit the "source" button.

Let's read in some data from a file.

```
comp.dat <- read.csv("compensation.csv")
```

If we add that to our script and source it you can see that comp.dat shows up in the workspace. Also we can type `ls()` in the console to see if its there.

Make sure your data is correct

NEVER TRUST ANYONE! ALWAYS CHECK YOUR DATA.

There are a few quick functions that we can utilize to make sure the data is all there and in the proper format.

```
names() # this will give you the variable name in the data
head() # this will print the first few lines of the data
dim() # gives you rows and columns the dimensions
str() # this will give you the structure of the data
```

Add the following lines to your script and source that file:

```
names(comp.dat)
head(comp.dat)
dim(comp.dat)
str(comp.dat)
```

Once a data frame is uploaded. You will want to attach it with the function `attach().` This will make all the column vectors accessible by name.

```
attach(comp.dat)
```

Now in the console our variables are accessible directly by name.

```
> Fruit
```

Another great R function is **`summary().`** This works on many different types of objects and can be a great way to get a quick handle on your dataset.

```
> summary(comp.dat)
```

Task

There is a file in R_files called islanddata.txt. This is a tab-delimited file type import this file and save it to an object called island.dat. Review the help for **`read.table()`** if necessary.

How many variables are there?

What are their names?

Did you get errors? What are they?

If the names are V1-V4 you need to alter your **`read.table()`** command to use the existing headers.

Data Manipulation Functions

Here is a list of basic data manipulation functions you will probably want to use in processing your data:

Script command	What it does
NA	Code for missing data
names(Example1)	Gives the names of variables in a data frame
comp.dat$Fruit	Indicates the variable *weight* in the data frame *Example1*

Subsets of vectors	What it does
Example1$weight[1]	shows the 1st weight value
Example1$weight[-1]	shows every weight except the 1st
Example1$weight[5:10]	shows the 5th to 10th weight values

Subsets of data frames [row, column]	What it does
Example1[1,]	shows the first row
Example1[,2]	shows the first column
Example 1[, c("weight", "tmt")]	shows the columns labeled weight and tmt
df$column	Shows the column listed in the data frame df (frame must be attached for this to work)
Transforming data	**What it does**
lweight<-log10(weight)	Takes the logs of weight to create a new variable lweight
sqweight<-sqrt(weight)	Square roots of weight become the variable sqweight
aweight<-asin(sqrt(weight/100))	Arcsine transform of the weight variable

Summary statistics	What it does
tapply(Y,A, mean)	Mean of vector Y at each level of factor A
tapply(Y,list(A,B), mean)	Mean of vector Y at each combination of factors A and B
tapply(Y,A,length)	Number of R observations at each level of factor A

Now that the data is attached, we can see a one of our data vectors by calling it by its name like this:

```
> Face
 [1] North North North North North South South South South South
West  West  West
[14] West  West  North North North North North South South South
South South West
[27] West  West  West  West  North North North North North South
South South South
[40] South West  West  West  West  West
Levels: North South West
```

You can also try some of the other options above.

```
>island$Face
>island[,2]
```

This will give you the second row.

```
>island[2,]
```

This will give you the 4-8 rows and columns

```
>island[4:8,4:8]
```

These easy subsets of data can allow you to quickly make new data frames that only contain the data you are interested in for that analysis session. Now, the most valuable and time-saving reason to use R is the quick sub-setting of your data by a factor or value. The function **`subset()`** can help you do this. For data frames, the subset argument works on the rows. Note

that subset will be evaluated in the data frame, so columns can be referred to (by name) as variables in the expression (see the examples).

```
> subset(island, Face == "North" & Island=”SanJaun”)
```
Note when telling R the value or description to include for the column Face, you need a two equal signs ==. This is true of numeric values also.

Like:
```
> subset(island, AlgalDensity == 47.59)
```

#You can also define a range of values like this:
```
> subset(island, AlgalDensity > 50)
```

This can also be accomplished like this:
```
> island[island$AlgalDensity > 50,]
```

Logical Operators	What it means
==	equal to
&	and
|	or
!	not
!=	not equal to

Task

Make a subset of the comp.dat that includes only grazed fruits with an initial root diameter less than 8.96. Save this as an object called my.subset. Add this command to your script and source the file when you are done. (note you will lose island.dat when you do this)

How many rows do you have in this subset?

What was your command?

How did you determine the number of rows?

Summarizing Data

A very super important time saving data manipulation procedure you will want to do is to summarize your data. Essentially, you want to make a summary table from your raw data table. One way to do this is to apply the **`aggregate()`** and **`tapply()`** functions.

Lets start by reviewing the help for **`aggregrate():`**

```
>?aggregate
```

Now we can apply this to our comp.dat:

```
> aggregate(comp.dat$Fruit, list(comp.dat$Grazing), mean)
```

This tells R to get the Fruit column then break it apart based upon the Grazing column and return the mean.

`tapply()` works similarly but instead of returning a data frame it returns a matrix:

```
> tapply(comp.dat$Fruit, list(comp.dat$Grazing), mean)
```

If you want to keep either of these values you must use **`<-`** and direct them to a new object.

The third position of either argument can take a variety of R functions (including things you create yourself) such as **`mean`**, **`sd`**, and **`median`** among many others.

Task

Fill in the chart on page 37 of your book "Getting Started with R" see how many you can fill in without looking them up.

Day 2 Websites

Bioconductor website

https://www.bioconductor.org/

Bioconductor Introduction

https://goo.gl/5Rgfqo