Biol 206/306 – Advanced Biostatistics
Lab 1 – An Introduction to R
Fall 2016

By Philip J. Bergmann

0. Laboratory Objectives

1. Find out about the characteristics of the R environment
2. Install R, and learn how to install packages you may need
3. Learn what a command prompt is
4. Learn what the menus in R allow you to do
5. Learn the syntax of R
6. Explore some basic operations and functions in R
7. Review some basic statistical tests

0.1 A few tips on using this lab manual

The format of this lab manual will be consistent between labs, and we will use one lab handout per lab session. Each handout will start with a **Laboratory Objectives** section, which will briefly list the big concepts that you should take away from each lab. This will be followed by a series of sections that describe the week's activities in more detail, including some limited background information. You should also refer to class discussions of primary literature and lectures given earlier in the week for more details, and you should also supplement your background knowledge with readings from other books, published papers, and the web, as you will need to understand concepts fully. Key words/new terminology will be provided in **bold** script, specific instructions on what to do will be ***bold and italicized***, and any sample code that you may need to use will be in `Courier New` font. Each lab handout will also contain pages with activities that you will need to complete and hand in for a grade.

1. What is R?

R is a computational environment that has sophisticated capabilities for data manipulation and storage, calculation (especially matrix algebra), graphing, and programming. R is a coding language, with many programmed **"functions"** available for use. Because it is a coding language, it is very flexible, allowing users to program virtually any sort of computational analysis they need. Many students find this flexibility and the use of code to do analyses intimidating. However, once you are accustomed to how it works, it is easy to implement very sophisticated analyses very efficiently. In the last ten years, R has become the environment of choice for people working in computational biology because it is open source, and can be used to implement analyses using statistics, mathematical modeling, and simulations. Because we will be learning many disparate statistical techniques, some of which are not implemented in standard statistical software, R is the natural choice, allowing us to avoid learning three or four different software packages that have narrower capabilities.

R is a type of **object-oriented programming** language. This differs from some older forms of programming that consisted of a series of written commands, sometimes indexed by line numbers. Instead, object-oriented programming is based on the manipulation of **objects** to carry out tasks. An **object** is any sort of data structure; it typically has a user-defined name and contains a wide possible variety of things. An object belongs to a **class**, which simply defines the type of object it is. The following is a list of classes to which most objects you will run across will belong:

a. **Vector** – A linear sequence of data.
b. **Factor** – A vector with a specific number of levels, or options that each datum can take (e.g., "male" and "female"). Factors can be ordered or unordered.
c. **Matrix** – A 2-dimensional sequence of data with a certain number of rows and columns.
d. **Array** – A sequence of data with three or more dimensions.
e. **Dataframe** – Superficially similar to a matrix, but more of a table, where columns and rows have names. This is closest to what you may think of with a spreadsheet.
f. **List** – An ordered sequence of objects, known as components. The components of a list can be objects of various classes.
g. **Function** – Might also be thought of as a command. Functions do various operations and manipulations of objects. This is often where programming in R comes into play.
h. Other classes – One can define any sort of class that one needs to do various analyses. While making our own classes of objects is beyond the scope of this course, we will be using special classes of objects, such as phylogenies, with are mostly of class *phylo*.

Objects also have **attributes**, which are parameters that define the characteristics of each particular object. The **mode** of an object is an attribute that describes the type of data that fills the object's elements. Mode can be **numeric** (when the elements are numbers), **logical** (true/false), **character** (letters instead of numbers), or **complex** (various other options). The **length** of an object defines how many elements are present in an object. For example, a vector consisting of a series of twelve True/False values would have mode=logical and length=12. The **dim** is an attribute that defines the dimensions of an object. For example, a 4x5 matrix of numbers would have mode=numeric, length=20, and dim=4, 5. There are many other possible attributes to various classes of objects, but these are the most common and important.

2. Installing R and some associated packages

The CRAN website

The official R website is the Comprehensive R Archive Network, http://cran.r-project.org/. It contains downloads for R for all platforms (Windows, Max OS, Linux), and all the official **packages** that people have written to do various analyses. A package contains a series of functions that do various tasks. It is worth checking this website out because it contains a lot of great information, including various free manuals. ***If you click on the "Manuals" link on the left***, you will see some of these manuals. "An Introduction to R" is recommended reading for anyone starting out in R. Although some of this lab goes over the same topics as the manual, "An Introduction to R" contains a more formal and thorough introduction to the environment.

Click on "Packages", also on the website's left side, and then click on "Table of available packages…" to access a listing of packages that are available for download. These packages

contain functions that have been written by various people to carry out various analyses. There are thousands of packages, and so, most analyses that you might conceive of doing already are fully or partly implemented in R, from the basics to the very advanced. ***Choose a package that sounds interesting and click on its link.*** You get some basic information about the package, and links to download both the package and a PDF reference manual that describes its functions. "Depends:" lists any packages that the package of interest requires to run. These have to be installed as well.

Click on the link to the reference manual PDF for the package that caught your eye. Each reference manual is formatted in the same way, starting with some basic information about the package and a table of contents, followed by a function-by-function account of the functions contained in the package. Sometimes, there is an account of the package itself or sample data used in the package, in addition to the actual functions. ***Scroll down in the reference manual until you come across the description of a function that does some sort of analysis***. The function name and title appear between the two horizontal rules. The **Description** describes what the function does, often in plain language. **Usage** gives you the syntax for using the function (see below). **Arguments** are the various parameters that you specify as input when using the function. **Value** is what a function returns as output. This is followed by the **Author(s)** of the function, **References** relevant to the function, and related functions that you may want to **See Also**. Finally, under the **Examples** heading there may be some examples of code that uses the function. The clarity and detail of function accounts varies widely because they were written by different people.

R Installation

In this course, we will be using the base R version. Some people use various packages for R that provide a basic **graphical user interface (GUI).** For example, Deducer and JGR provide a nice GUI for generating graphics, and many people use R Studio, which organizes the workspace nicely. You ae free to use these packages if you like, but note that the labs are written for base R, and the instructions do not support these various GUIs. In addition, JGR sometimes has compatibility issues with both R and Java, and it cannot be used to load some of the data that we will use in this course, like phylogenetic trees. It is recommended that students use base R or R Studio instead.

1. ***Go to the Comprehensive R Archive Network (CRAN) website: https://cran.r-project.org.***
2. ***Under "Download and Install R" click on the link for the OS that you are using.***
3. ***For Windows, click on "install R for the first time", the on "Download R 3.3.1 for Windows. Follow the installation instructions.***
4. ***For Mac, click on the link for "R-3.3.1.pkg", and follow the instructions.***

You should now have a shortcut to R on your desktop. You may have two short cuts – one for the 32-bit version of R and one for the 64-bit version. You can use either version. However, modern computers have a 64-bit architecture, so that version provides enhanced performance.

Installing and loading packages in R

You can download packages that you need in R, as opposed to from the R website. This has the advantage that other packages that your needed package depends on are automatically

downloaded and installed as well. We will install two packages that you will need later in the semester to see how this works. The packages are **geiger** and **phytools –** both are for doing phylogenentically-informed statistics and analyses. They depend on other packages as well, so notice that those also get downloaded. Do the following:

1. ***Open R on your computer.***
2. ***If you are using a PC: Go to the "Packages" menu and select "Install Package(s)…" item. In the future, an instruction such as this will simply be presented as "Packages> Install Package(s)…"***
   - If you are using a Mac: Packages & Data> Package Installer. The click on the button "Get List", check off the box "Install Dependencies", and select the package(s) you would like to install.
3. ***A dialog may prompt you to "Select a CRAN Mirror". The packages are downloadable from many servers located in many places in the world. You can select any mirror – it doesn't really matter, although occasionally one may not have the full list of packages – the mirrors in California typically work well.***
4. ***Select the packages that you need. Select both geiger and phytools. On a Windows platform, if you hold down the CTRL key while selecting options, you can select more than one at a time. On a Mac, use the Command key.***
5. ***Click OK.***

Before using a package, it must be both installed and loaded. Installation is a one-time process – you will not have to install geiger or phytools again. It involves downloading and installation. Loading packages has to be done each time that you open R and decide that you need them. Loading a package makes it available for use during an R session. In R, do the following:

1. ***If you are using a PC: Packages> Load Package…, then select geiger.***
2. ***If you are using a Mac: Packages & Data> Package Manager, the check off the boxes beside the packages you would like to load. Make sure that "not loaded" changes to "loaded".***

Notice that R loads not only geiger, but a series of other packages that geiger depends on to work.

3. What is a command prompt?

R uses a **command prompt** to interface with the user (you), which is very different from how most people are accustomed to interfacing with their computers. Most people are accustomed to a GUI, generally involving a series of windows with menus and buttons that they can click on with the mouse. A command prompt is simply a symbol to the right of which you type commands to control the software. In the days of yore, PCs ran on DOS (disk operating system), which was command prompt based. To this day, this is an option (although much reduced in capability) on PCs. In Windows, ***type "cmd" in the "Search Programs and Files" area in the Start Menu***. On a Mac, ***click on the magnifying glass icon for spotlight search, and type in "Terminal".*** A dialog pops up where you can interface with your computer more directly. This prompt includes a description of what folder you are in.

What does the command prompt look like when you do this?

Likewise, R uses a command prompt. ***Open R. What does its prompt look like?***

At the prompt, type "citation()". This provides information on how to cite R when you use it for research. Some people find the command prompt intimidating because they cannot simply click on things and see what happens. Work to get comfortable with the command prompt – it is your friend!

4. R Menus

Like most modern software, R has a set of menus. However, they are relatively basic because R is prompt-based. Here, we will introduce only some select options available in R's menus.

1. PC: **File> Load/Save Workspace…**
   Mac: **Workspace> Load/Save Workspace**
   Allows you to save all the objects in R memory during a session. Loading them allows you to continue your work. The workspace is saved as a .RData file.
2. PC: **File> Load/Save History…**
   Mac: **File> Load/Save**
   Allows you to save or load everything that you have typed to the command prompt as a .Rhistory file. You can also open this file in notepad and revisit exactly what commands you used to do an analysis.
3. PC: **File> Change Dir…**
   Mac: **Misc> Change Working Directory**
   Allows you to change the active directory. This is the folder that R goes to to save or load files. When starting a session, it is good practice to change the directory to the one in which the files for your project are located.
4. PC: **Misc> Stop current/all computations**
   Mac: **Edit> Complete**
   This allows you to stop R if you have just started an analysis that is time consuming and realized that it isn't the analysis that you want to do, or if you program an infinite loop.
5. PC: **Misc> Remove all objects**
   Mac: **Workspace> Clear Workspace**
   Sometimes you have many objects in R's memory that you do not want. This removes everything and you can start with a clean slate.

It is always a good idea to save your R workspace and your command history when doing an analysis (or lab). This documents what you have done and allows you to both recreate it and continue from where you left off.

5. R syntax

Syntax is the study of the rules of forming sentences in a given language. This applies both to linguistics and computer languages. Every coding language has a syntax that a programmer

must use to successfully use implemented functions. The basic syntax in R can be represented as:

> function(argument1,argument2,…)

Where ">" is the command prompt, a function is essentially to command given by the user, and arguments are the input that the function needs, enclosed in parentheses. Arguments are separated by commas. Expanding on this, you will often use the following syntax:

> object <- function(arguments)

Where, you store the output of the function in an object that you name. <- is an arrow that assigns the output to the object. You can either use <-, as above, or use -> and put the object name at the other end of the line – there's no difference.

Arguments of mode="character" are enclosed in quotation marks. This syntax is relatively simple, but pay close attention to it, forgetting a parenthesis, quote, or comma will lead to an ambiguous error statement and can result in lots of wasted time trying to figure out what is wrong. If something doesn't work, the first thing to do is to make sure that your syntax is correct. You will get introduced to more syntax details and learn how it works as you learn R.

6. Some Basic Operations and Functions

Now that you are familiar with what R looks like and how the syntax works, we will spend some time going over some useful operations and functions in the environment. We'll start with some basic housekeeping functions, then move on to actual data.

> ls() – Lists all objects in memory. These are the objects you have at your disposal.
> rm(object) – Removes the specified object from memory.
> object – Prints out the object to the screen.
> str(object) – Gives all the attributes of an object. This is different from the object itself.
> help(function) – Opens a web browser with documentation for specified function.
> ? function – Same as help.
> # stuff – The # sign identifies a line as a comment line that is not read by R. Everything between the # and the next carriage return is a comment. This is useful when writing scripts or functions of your own because you can document what each line of code does.
> library(package) – Loads a package. You can do this using the menus as well.

Operations

Operations are simple procedures that get applied to objects to do calculations of various sorts. R can be used as a calculator, and so, you can do calculations using +, -, *, /, ^, log(x), exp(x), min(x), max(x), sin(x), cos(x), tan(x), sqrt(x), where **x** is a number or a more complex object. In addition, there are operations that act on larger objects with multiple elements (for example, matrices):

> range(x) – Returns a vector of two numbers, min and max.
> mean(x) – Returns the mean of the elements.
> var(x) – Returns the variance.
> sum(x) – Returns the sum.
> length(x) – Returns a number describing the number of elements in **x**.

`> c(x,y,z)` – concatenates multiple objects into one. This is very useful.
`> x:y` – Produces a sequence of numbers from x to y, inclusive.
`> sort(x)` – Sorts the numbers in a vector from lowest to highest.

Try out these functions. Start by making a vector of numbers as follows:
```
> v <- c(2,5,7,3,1,9,4,8,6)
```

At the prompt, type "v", then try "str(v)". How do these two ways of viewing the object differ? What information does the latter tell you?

For vector v, what is the range?
The mean?
The variance?
The length?
The mode (not numeric)?

Now type "sort(v)". How would you store the sorted vector in an object called 'sv'?

What is another way you produce a vector called 'sv' with the same contents without just typing in all the numbers in order? Think about doing this efficiently! Often in R, and programming in general, there are multiple ways to do the same thing.

You can also combine operations. This can be as simple as something like "2*mean(v)" to more interesting things like "twos <- 2*1:5". What does the latter give you?

For more flexible sequences, you can use the seq and rep functions:
```
> seq(from, to, by, length, along)
> rep(x, times=a, each=b)
```
Here, 'from' is the start of a sequence, 'to' is the end, 'by' is the interval, 'length' is how many elements the sequence runs for (this cannot be used in conjunction with 'to'), and 'along' can only be used alone and numbers the elements of a specified vector. Furthermore, 'times' is the number of times **x** is repeated end to end, while 'each' is the number of times a vector's element is repeated before the calculation moves on to the next element.

Try these commands out. Create a vector that is the same as "twos" using the seq function. Then try to following:
```
> rep(twos, times=4) -> two_by_four
> rep(twos, each=4) -> two_four_times
```
How does the output from these two lines differ?

The functions **cbind(x,y,…)** and **rbind(x,y,…)** bind two or more vectors into a matrix, where each vector becomes either a column or a row, respectively. ***Bind the two vectors you produced above into a single matrix. Use the cbind function, assigning the output to an object called 'col2', and then use the rbind function, assigning the output to an object called 'row2'.***

What is the length of col2?

Typing '**dim(col2)'** will give you the dimensions of the matrix. ***What are the dimensions of 'col2' and 'row2'? How about of 'v'?***

You can call on a particular element, row, or column in a vector or matrix using square brackets.
Try the following. What does each line of script return?

```
> v[4]
> col2[5,1]
>col2[5,]
>col2[,1]
```

You can also turn a vector into a matrix by imposing dimensions on it. ***Try the following:***

```
>dim(two_by_four) <- c(5,4)
```

Take a look at the new two_by_four. How does it differ from the old one?

Just to more clearly see how all of this works, using the new two_by_four object, replace the element in the third row, second column with the number zero. What did you type to do this?

The last thing that we will do with the current set of objects is modify the two_by_four matrix to have its rows and columns labeled with names, which sometimes helps make viewing of the data easier. We will start by renaming the matrix so that there is less typing. ***Try the following:***

```
> two_by_four -> tf
> rn <- c("a","b","c","d","e")
> cn <- c("one","two","three","four")
> rownames(tf) <- rn
> colnames(tf) <- cn
```

Please do not just copy and paste the code from the lab into R, as that prevents you from learning effectively. Although it takes a little more time, type in everything to the command prompt. This ensures that you will learn the material and be ready for subsequent labs.
Now view the matrix to confirm that rows and columns are labeled as expected. If you type "str(tf)", you will now see that the row and column names are stored there under the attribute "dimnames".

Importing and Manipulating Data

We will continue our introduction to R and other object classes by importing and working with some real data. Most things in R can be accomplished multiple ways, and it is often most efficient to start by learning one convenient way of doing something and then sticking with it. If situations arise when another approach is necessary, one can quickly learn how to modify their approach. Importing data is one of these tasks. One can import data saved as tab-delimited text files, comma-delimited files, or others. We will view a dataset in Excel, save it as a tab-delimited text file, and import that into R. You can then use this approach to import any of the datasets that we work with during the semester.

Open Microsoft Excel, then open the file "Lab01 Data.xlsx".

The spreadsheet contains two worksheets: one is a readme that describes what the variables are. The other is the actual dataset. Take a look at both. ***What do the variables 'SL' and 'nP' stand for?***

Notice that the worksheet with the data contains six columns, but only five of them have headings. The column without a heading is the leftmost column and it contains a unique fish identifier. Omitting the heading for this leftmost column allows R to automatically identify it as rownames, saving some steps in setting rownames. It is important that each row name is unique. As you look at the data worksheet, also notice that there is no formatting (no bolds, underlines, shading), and that all of the column headings are single words (no spaces). A lack of formatting isn't very important because it will be discarded when the file is saved as a text file (but consider that fancy formatting is often a waste of time – focus on the data in the file). Column headings lacking spaces is important because it makes importation into other software much easier. Consider never having spaces (or special characters) in your column headings.

The reason that we start in Excel here is because Excel is very powerful at manipulating data and doing some basic calculations. It is suggested that you do data manipulation, like sorting, adding/calculating new columns, etc. in Excel, prior to saving as a text file. Let's practice some useful Excel functions:

1. ***Start by highlighting the whole dataset and sorting the data by sex, then SL (Data> Sort…).***
2. ***Now, so that we can compare the relative mass of parasites in a fish, let's make a new variable, called rPmass (relative parasite mass). Make the heading immediately right of the existing columns.***
3. ***In the first cell below your new heading, type "=F2/D2", or click on the appropriate cells, as appropriate.***
4. ***Fill down your formula to complete the column, and leave all of the cells in the column highlighted.***
5. ***Finally, it is important to replace the formula with the calculated numbers. If you leave the formulae, then whenever you change a Mass or MassP, the rPmass value will be updated. This is often a desirable property, but formulae do not import well into other software. So, Copy the cells (CTRL+c), and then Paste Values (or do "Paste Special…> Values", depending on your version of Excel). Notice that the formulae have been replaced by the numbers.***

The next step is saving the dataset as a tab-delimited file:

1. ***Go to "File> Save As…"***
2. ***Under "Save as type", select "Text (Tab delimited) (*.txt)"***
3. ***Type in a filename, e.g., "Lab01 Data", and click "save"***
4. ***Finally, follow through the dialog boxes with "OK" and "Yes", but then do not save over the xlsx file.***

Now you are ready to import the saved text file into R. ***Start by setting the active folder by going to "File> Change dir…", and navigating to the right folder (which will probably be under: C:\Users\<your login name>\..."). If using a Mac, you will find this in "Misc> Change Working Directory". Then, at the command prompt, type:***

```
> data <- read.table("Lab01 Data.txt", header=TRUE)
```

For the line above, label what the function is, what its arguments are and what object is that you are importing the data to. "header=TRUE" is a logical statement informing R as to whether the file has column headings or not. If TRUE, then colnames will be filled in automatically. Since we omitted a heading for the fish ID column, this information will automatically make up the rownames. When using logical operators like TRUE and FALSE, you can also just type T and F. Note that whether you type the whole word or just the first letter, it needs to be in capitals.

View what you imported, then also use the str() function to see its characteristics.
Notice that the data are imported as a data.frame, not a matrix. This is a new object class for you, one that is used very widely for data such as this. Each column is listed, giving you its name, its mode, and then some of the first data in each. Notice that "Sex" is a factor with two levels, which is what we want. You can also view a summary of the data with:

```
> summary(data)
```

What is the mode (not numerical) of the variable nP?

How many males and how many females are in the dataset?

What is the mean parasite mass for all fishes?

Looking back to the str(data) output, note that this can be very useful because you get a concise listing of all variables and what sorts of variables they are. It is suggested that any time you import data into R, you view the object by typing its name at the prompt AND you also use str to view its structure. By doing this, you can make sure that the data were imported the way you wanted prior to doing any analysis or manipulation of the data. In particular, make sure that components of objects, like columns in a dataframe are of the mode that you need. It is common to need to convert a variable to a different mode to do what you need. For example, if you have sex represented as zeros and ones in a dataset, they will be imported as integers instead of as a factor. It is the latter that you want!

You can also refer to a specific variable in a data frame very easily using the $ symbol. Try it:

```
> data$Sex
> sex <- data$Sex      # Allows you to assign the Sex variable to its own object.
> less_data <- data.frame(sex,data$SL,data$Mass)          # Allows you
```

to make an object that is a data frame containing a subset of variables, or combining elements of different objects. Be careful that these objects are all the same length and in the correct order.

In the last line of script, notice that the rownames disappeared and that the colnames changed to tell you where the variables came from. ***If you want to restore the col and row names from the*** **data** ***object, do the following:***

```
> rownames(less_data) <- rownames(data)
> colnames(less_data) <- c("Sex","SL","Mass")
```

Now take a look at the list of objects you have in memory, just to see what you've produced.

Plotting Data

Our next step in exploring R is to see how to plot data. R has a wide range of plotting functions, and plots can be done from the command line, just like everything else. ***For example, try:***

```
> boxplot(data$SL)
```

A boxplot appears for standard length for all the fish in the dataset. ***To get back to the prompt, click the "Return Focus to Console" button in the upper left, under the menus.***

Another way to represent a distribution of data, like SL, is using a histogram. ***Do this with the function:***

```
> hist(data$SL)
```

Since the graphics window is already open from our boxplot, you can view your new plot by going through the menus: Windows> 2 R Graphics… Switch back to the console after viewing your plot.

You can change the number of bins using in the histogram to get coarser or finer resolution using the "breaks" argument. ***Try:***

```
> hist(data$SL, breaks=20)
```

Try several different numbers for "breaks" to see how it works.

Thinking back to the boxplot, it is often useful to compare boxplots for multiple groups. If you have a categorical variable and you want a boxplot for each, ***you can do this easily as follows:***

```
> boxplot(data$SL~data$Sex)
```

Note that SL is a continuous variable in all of these cases and that Sex is a factor.

Another very common plot in science is the scatter plot, used when you have two continuous variables. ***You can make a basic scatter plot by typing:***

```
> plot(data$SL~data$Mass)
```

Notice that both variables are continuous, and that the x-variable is to the right of the tilde (~), while the y-variable is to the left. Be careful with this because plot can be used in two forms:

```
> plot(y~x)
> plot(x,y)
```

The order of your variables is important in relation to the notation you use. In R, the tilde signifies a function and the response is always on the left.

In some situations, you may want to add a regression line to your scatter plot. You can do this additively by first making the plot above and then adding the line. Make sure that your graphics window has the scatterplot shown above. ***Then type:***

```
> abline(lm(SL~Mass))
```

You can also make the line a different color with an extra argument:

```
> abline(lm(SL~Mass),col="red")
```

Notice that since the setting for the col argument is a word, it has to be in quotes.

As we did with the boxplot, you may want to use different symbols or colors to represent the two sexes (or different categories in general), and include a regression for each. ***To do this, first load the package "car". It comes automatically with R, so you don't have to install it, but you do have to load it. Then use the scatterplot function:***

```
> scatterplot(y~x|z, smoother=F)
```

Here, x and y are defined as above, and z is a categorical variables. The smoother argument suppresses lines that are moving averages of the data. ***If you do not want the regression line plotted, add the argument: `"reg.line=F"`.***

The plots described above should accomplish much of what you need to do in this course and in most situations. There are a number of arguments that are also useful in tweaking the appearance of your graph. ***Try adding the following arguments to your scatterplot function and write down what each one does:***

```
xlab="<label>"

ylab="<label>"

cex=<number>

cex.lab=<number>

legend.plot=T|F
```

For the above, note that things in < > are what you fill in (don't include the pointy brackets), quotation marks must be written and indicate a non-number (letters or combinations of letters and numbers), and T|F indicates that you choose to set an argument as TRUE or FALSE, and the first option written is the default (if you want the default, you don't have to type the argument).

Doing Calculations and Analyses Using a Dataset

The final exercise using R today will allow you to bring together everything that you've learned about (and learn a few more functions) in a real-world statistical problem. You should all be familiar with the t-test, and we will use this to compare relative parasite mass between males and females. Since there is an equal number of males and females in the dataset, we can use a simplified two-sample t-test for equal sample sizes. The equation for the test statistic is as follows, with m and f referring to the two sexes in this example, n being the sample size for one sex, and s^2 being the variance:

$$t = \frac{\bar{Y}_m - \bar{Y}_f}{\sqrt{\frac{s_m^2 + s_f^2}{n}}}$$

Now let's consider how we can calculate *t* from our data frame. First, we need to introduce one new function that is simple but very useful for programming:

```
> tapply(vector, factor, function)
```

The tapply function applies a specified function to a vector that is grouped by a factor. In this case, our vector is **rPmass** (it is a sequence of numbers of relative parasite mass), our factor is **Sex** (it can be male or female), and if we want to calculate the mean relative parasite mass for each sex, then our function is **mean**.

1. ***Make an object named "means" that contains the mean rPmass for the two sexes using the tapply function.***
2. ***We can use the tapply function to give us the sample sizes as well. In this case, the function is "length". Make an object named "ns" that contains these. A hint is that for this step you can simply keep the vector and factor the same as before.***
3. ***Again use the tapply function to give an object containing the variances, called "vars".***
4. ***Note that the tapply function outputs a vector (type "means" to view one), and so you can refer to a particular element in the vector using square brackets. Try this as well, by typing "means[1]" and then "means[2]".***
5. ***Now calculate "t" using what you have learned, calling the object containing the output "t". Remember that "n" is the sample size for one sex, since both sample sizes are equal.***

Now that you have the t statistic, you can calculate its probability from a built-in t-distribution in R. The function for the cumulative probability distribution for t is **pt(t,df)**, where **t** is your test statistic and **df** is the degrees of freedom for the test.

What are the degrees of freedom for a two-sample t-test with equal variances & sample sizes?

Since the **pt** function gives the cumulative probability distribution (CPD), which is sigmoidal in shape (see Figure 1 on the next page), the resulting probability will be one-tailed and will be too high if $t > 0$ because the CPD is asymmetrical, unlike the probability density function (PDF). Therefore, to get the correct p-value for a two-tailed test when $t > 0$, we need to subtract the value of **pt(t,df)** from one and multiply the result by two. S***ave the result as an object "p".***

Assignment (5 points)

A note about table formatting: When using a table to present quantitative data, there are several professional rules that are always followed. During this course, you should follow these rules as well because one of the course goals is to teach how to professionally present quantitative data. The rules you should follow are:

1. Use the same number of decimal places for each entry that contains the same kind of information. For example, in the table below, your means should have the same number of decimals, as should your variances, and your sample sizes. Some of these may use the same number of decimals, but some may not. For example, it might make sense to provide some number of decimal places for mean, but no decimals should be used for sample size. In statistics, it is the number of decimal places that is important, not number of significant digits. There is some flexibility in number of decimal places used, but it should be consistent with how many are needed to come to the conclusions you come to. Typically, p-values are presented to three or four decimal places.
2. Line up your decimals. If you have a column of numbers that all have the same number of decimals, left justify them so that the decimals line up. If they don't the table is hard to read.
3. Tables should have headings, and those headings should always go above the table. When making a table in this course, please provide a descriptive heading. Also note that figures have captions, not headings, and these go below the figure, not above.

Complete the following table.

Parameter	***Male***	***Female***
Mean		
Variance		
n		

How many degrees of freedom are there for this test?

What is t for this test?

What p-value did you get?

What did you type to get the t statistic?

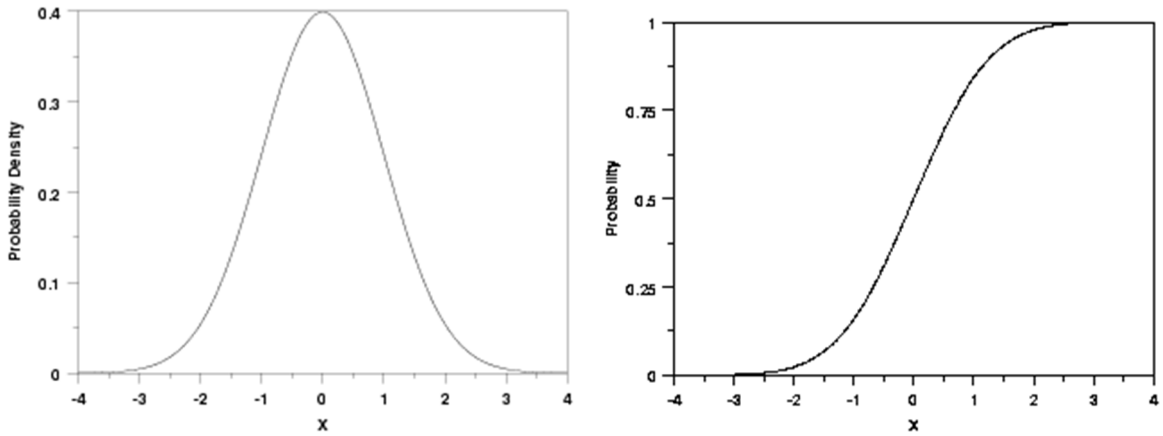

Figure 1: Probability density function (left) and cumulative density function for a normal distribution. A t-distribution would look very similarly. Note that the CDF results from a cumulative summing of the relative area under the curve of the PDF, such that the CDF has a maximum value of 1.

At this point, you might be thinking that R is a terrible idea for doing statistics if it takes so many calculations to do something as simple as a two-sample t-test. Keep in mind that this was an exercise to get you used to working in R and to show you how it can be very useful at manipulating data. Another huge benefit of R is that many techniques have been implemented in it, and users constantly make more functions to add techniques. Being an open source community, these functions are available to us for free. As it happens, various forms of the t-test have been implemented in R, and so a single function can be used to do the t-test we did above. Here is the function:

```
> t.test(formula, alternative=c("two.sided", "less", "greater"),
mu=0, paired=FALSE, var.equal=FALSE, conf.level=0.95)
```

- In **formula**, you can either specify a single vector if doing a one-sample t-test, or use the form **vector~factor** if doing a two-sample test.
- **Alternative="x"** is used to specify whether the test is two-tailed or one-tailed (two-tailed is default if you don't specify this, which is why it is listed first). Note that since these options are characters, they should be in quotes, but you don't need to include the c().
- **mu=0** is used to specify the value you are comparing a sample to in a single-sample t-test, or the expected difference in a two-sample t-test. The default is zero – a standard test, so that is what is listed in the syntax, above.
- **paired=FALSE** is default – a two-sample t-test. If TRUE, then you have a paired-sample t-test. Note that logical operators (TRUE/FALSE) must be in all caps or can be abbreviated with **T/F**. Logical operators should not appear in quotes.
- **var.equal=** is a logical operator informing whether the variance of your two samples is equal. If FALSE, then the calculation of the test statistic takes this into account.
- **conf.level=** is the confidence interval (95% by default) of the difference between treatments.

Try out the t-test function. If your desired option for an argument is the default, then you can omit it from your line of script. Note that you should simply assume that variance is equal between sexes, something we assumed in our manual calculations. Do you get the same numbers as from the test you did manually?

What are the 95% confidence intervals that you get from the t-test?

A Review of Some Basic Statistical Techniques

The final exercise of this lab is to re-familiarize yourself with some basic statistical techniques. Most of this should be review from your introductory statistics class. However, if you are not familiar with any techniques, research them in textbooks and online to obtain the information that you need. Complete the table on the next page. The entry for the two-sample t-test is completed to show you what is expected.

Assignment (5 points): Summary of characteristics for some basic statistical techniques

Test	Test Stat	Parametric/ Non-param	Tests Hypotheses About	# of Groups	Null Hypothesis Tested
One-sample t-test					
Two-sample t-test	t	P	Mean	2	The means of the two samples are not significantly different from one another.
Paired-sample t-test					
Analysis of Variance					
Kruskal-Wallis Test					
Bartlett Test					
Kolmogorov-Smirnov Test					
Bivariate Regression					
Pearson Correlation					
Spearman Correlation					
Chi-Square Test					