R for Fishery Biologists

ORCFWRU/ODFW Workshop

Summer 2016

Jim Peterson

Oregon Cooperative Fish and Wildlife Research Unit

Oregon State University

jt.peterson@oregonstate.edu

Contents

INTRODUCTION TO R

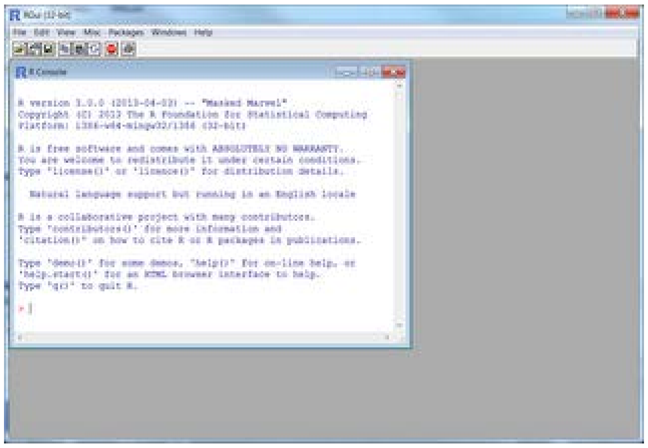
R is an open-source, objected-oriented language. When we say R is object-oriented, what we mean is that everything that R does or contains is based on objects. Objects in R include:

- numerical values letters or other symbols that stand for numerical values that can range from single values (often called scalars) to entire databases (defined in R as data frames)
- other structures, such as lists;
- functions, series of mathematical or statistical operations; and
- the results of statistical analyses (theses often can be confusing-- but not to worry).

For those of you experienced with other programming languages (e.g., C, Fortran, SAS), object orientation can initially be confusing. In this course, the structure of objects will be very simple and straightforward; as we get more familiar with R and get into more complex problems the power of the object-oriented approach will hopefully become clearer.

The R interface

Unless you are a total geek, you will more than likely be using a graphical user interface (GUI, pronounced--'gooey") to work in R. Fortunately, you have two options (ok, probably more than two but we will only discuss these two). The first comes with the standard R download from the Cran website. You can assess this by clicking on the "Rgui.exe" file in the "bin" folder that was created during the R install. In the computer lab, R should be listed in the program files list (hint: click the windows explorer icon). Open the standard R gui and take a look. It should look something like this (you may have a different version though):



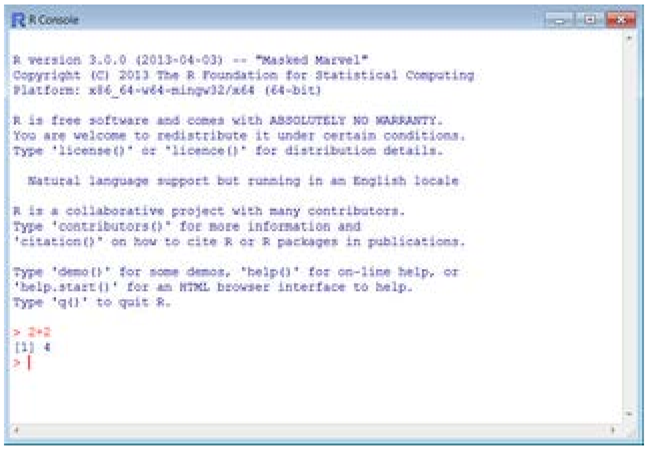
```
R version 3.0.0 (2013-04-03) -- "Masked Marvel"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: i386-w64-mingw32/i386 (32-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

It's fairly bare bones, but quite useful. The other alternative to this GUI is Rstudio, which also is open source and freely available (more below). Both interfaces can do roughly the same things though Rstudio has some additional functions that we think that you-- as nubies-- will prefer (JP definitely prefers Rstudio). Everything we cover in this course can be performed using the standard R GUI or Rstudio with the exception of a few procedures that we will cover in this lesson. Before we go over Rstudio, let's explore the standard GUI first. Above, shows the R console, which displays the results of analysis and messages associated with any code that is either entered in the command line (after the red arrow "**>**") or using something called a script. A script is a file that contains a bunch of R code that you can save and use over and over again. We will mostly be using scripts throughout the course, but for now let's just use the command line.

We assume that you all know the mathematical operators that we will be using in the course but just in case:

Operator	**Description**	**Example**
+	addition	x plus y is **x + y**
-	subtraction	x minus y is **x - y**
*	multiplication	product of x and y is **x*y**
/	division	dividing y by x is **y/x**
^ or **	exponentiation	y raised to the x power is **y^x** or **y**x**

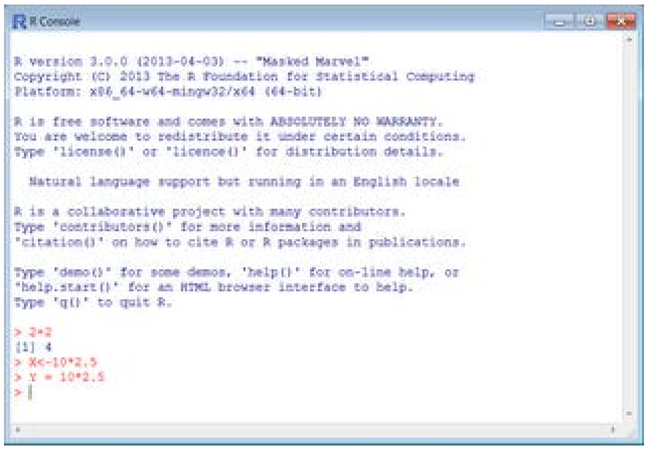
As we mentioned, you can enter commands directly in the R console. For example type in 2+2 and hit return. You should get something like that below.



```
R version 3.0.0 (2013-04-03) -- "Masked Marvel"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

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

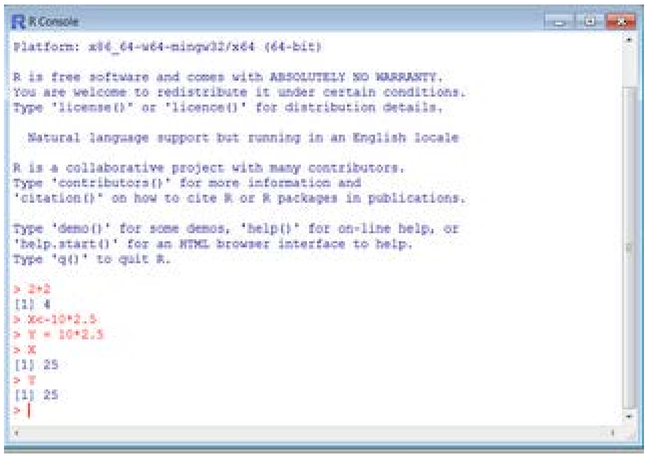
You can see the command that we typed in before the arrow "2+2" and the result below, 4. Try a few more commands for grins-- we're computing! Notice that the command console gave us the answer right after we submitted the command.

What about if we want to save the result of the operation? Well, we need to assign the answer to an object (your first!). We do this by using as assignment operator, either an equal sign "**=**" or an arrow followed by a dash "**<-**". For many of the objects we will be using, these two assignment operators can be used interchangeably, BUT (warning warning!!) later on we will find that this may not work of all objects. Lets create an object that is the product (reminder: multiply) of 10 and 2.5. Remember the product operator is the asterisk "*****" and assign the value to an object X using the command console. Do the same for an object Y but use the other assignment operator. For example,



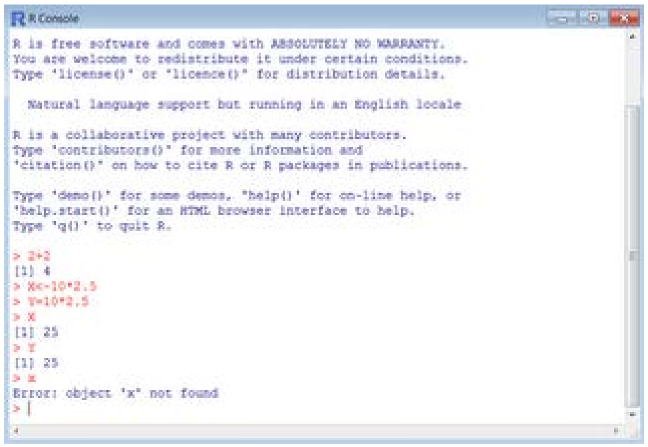
```
R version 3.0.0 (2013-04-03) -- "Masked Marvel"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 2+2
[1] 4
> X<-10*2.5
> Y = 10*2.5
> 
```

You should notice that the results of the operation were not printed in the console--that's because they were assigned to each object. If we want to see the result, we simply type Y or Y in the console and the results should be printed, For example,



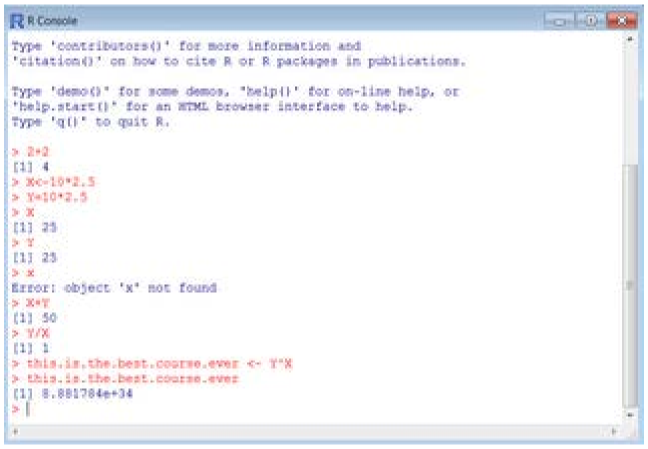
```
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 2+2
[1] 4
> X<-10*2.5
> Y = 10*2.5
> X
[1] 25
> Y
[1] 25
> 
```

Both values should be the same and they are. Objects are not restricted to single characters and can consist of combinations of numbers, letters, period, and underscore. Note that special characters (e.g., #, $, @, and commas) are used for other specific purposes, so stick with, "a,b,c,yx,dog,cat...", "1,2,3,55,19...", ".", and "_" when naming objects. For example, we could have used "jims.Y" or "jims_X" rather than X and Y. This makes it easier to keep track of data. For example, the object "fish.wt" could contain fish weights. *WARNING, WARNING object names are* ***CASE SENSITIVE***, i.e., typing lowercase x in the command window will result in the following error:



```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 2+2
[1] 4
> X<-10*2.5
> Y=10*2.5
> X
[1] 25
> Y
[1] 25
> x
Error: object 'x' not found
> |
```

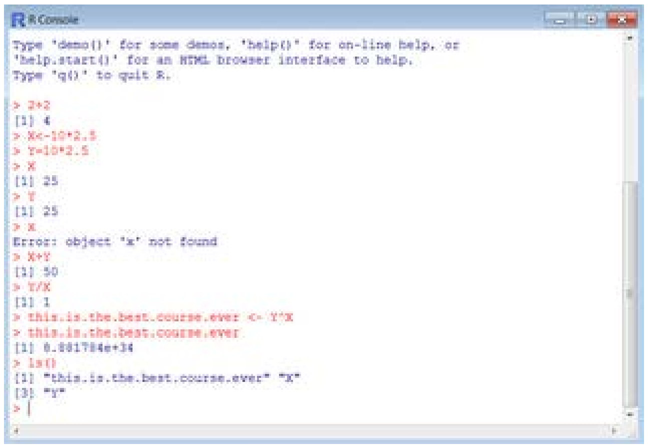
More often than not, forgetting this fact can lead to real headaches. Our suggestion: start now to establish a naming convention for yourself, such as only use lowercase for certain types of data and numbers for various versions of the object. As you might have guessed, we can perform operations of objects. To convince yourself, perform a few operations using X and Y, go ahead and assign the result of one of the operations to an object names "this.is.the.best.course.ever". For example,



```
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 2+2
[1] 4
> X<-10*2.5
> Y=10*2.5
> X
[1] 25
> Y
[1] 25
> x
Error: object 'x' not found
> X+Y
[1] 50
> Y/X
[1] 1
> this.is.the.best.course.ever <- Y^X
> this.is.the.best.course.ever
[1] 8.881784e+34
> |
```

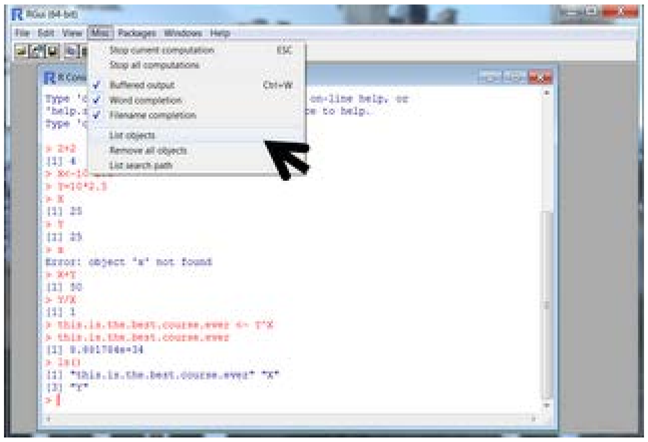
The neat thing about the console is that you can recall commands simply by using the up and down arrows on your keyboard. Go ahead and put the cursor in the command window and press the up arrow. The last command should reappear for the above example *this.is.the.best.course.ever* should reappear. You can continue to us the up and down arrows to scroll through your previous commands.

The objects that we created are stored in the ***working directory***. To list the contents of the working directory, we simply type "`ls()`" in the command window. For example,



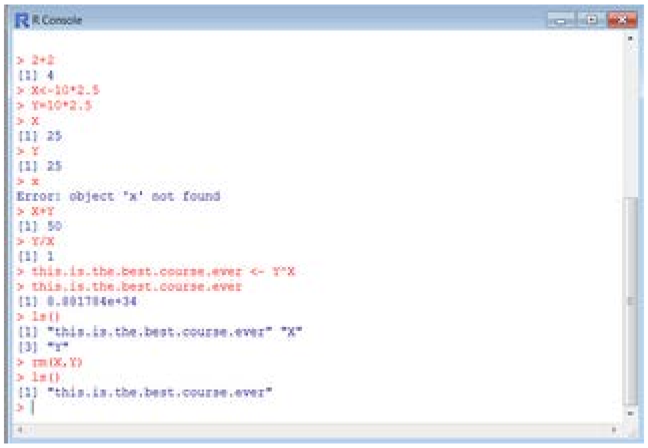
```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 2+2
[1] 4
> X<-10*2.5
> Y=10*2.5
> X
[1] 25
> Y
[1] 25
> x
Error: object 'x' not found
> X+Y
[1] 50
> Y/X
[1] 1
> this.is.the.best.course.ever <- Y^X
> this.is.the.best.course.ever
[1] 8.881784e+34
> ls()
[1] "this.is.the.best.course.ever" "X"
[3] "Y"
> |
```

Alternatively, we can go to the "Misc" menu at the top of the GUI and select "List objects":



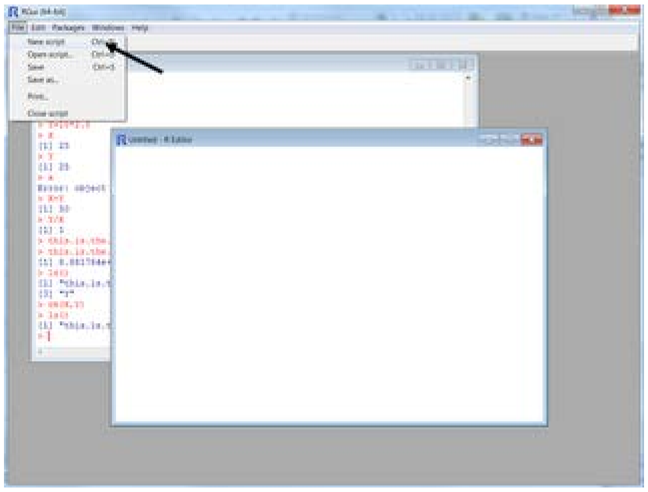
```
> 2+2
[1] 4
> X<-10*2.5
> Y=10*2.5
> X
[1] 25
> Y
[1] 25
> x
Error: object 'x' not found
> X+Y
[1] 50
> Y/X
[1] 1
> this.is.the.best.course.ever <- Y^X
> this.is.the.best.course.ever
[1] 8.881784e+34
> ls()
[1] "this.is.the.best.course.ever" "X"
[3] "Y"
>
```

For now, it may be easier to use the menu driven commands. Eventually, you will find that it will be better to learn and use the commands. We see here that it listed "X", "Y" and "this.is.the.best.course.ever". If we wanted to get rid of one or more of these objects, we use the remove command as "`remove(object name)`" or for short "`rm(object name)`" where "object name" is the name of the object you want to remove. If you want to remove more than one object, you separate the object names using a comma. Lets remove "X" and "Y" and list the contents of the working directory:



```
> 2+2
[1] 4
> X<-10*2.5
> Y=10*2.5
> X
[1] 25
> Y
[1] 25
> x
Error: object 'x' not found
> X+Y
[1] 50
> Y/X
[1] 1
> this.is.the.best.course.ever <- Y^X
> this.is.the.best.course.ever
[1] 8.881784e+34
> ls()
[1] "this.is.the.best.course.ever" "X"
[3] "Y"
> rm(X,Y)
> ls()
[1] "this.is.the.best.course.ever"
>
```

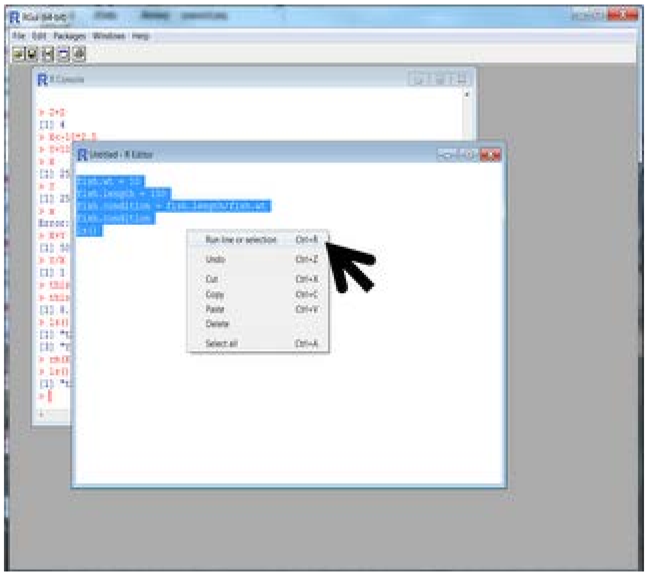
Not too shabby. Before we go any further, lets talk **R scripts**. An R script is a text file that contains R commands (and data if you want). Using a script is infinitely easier than using the console and typing in commands, so lets get started. In the standard R GUI, we click on "File" and select "New script" or use Ctrl-N:



The new script is shown in the window above. You can save and name the Script anyway you want-- that said, we strongly recommend using ".r" or ".R" as the file name extension. We will be using that convention throughout this course. To create a script, simply type the commands into the script and submit to R. In the script, type in the following commands

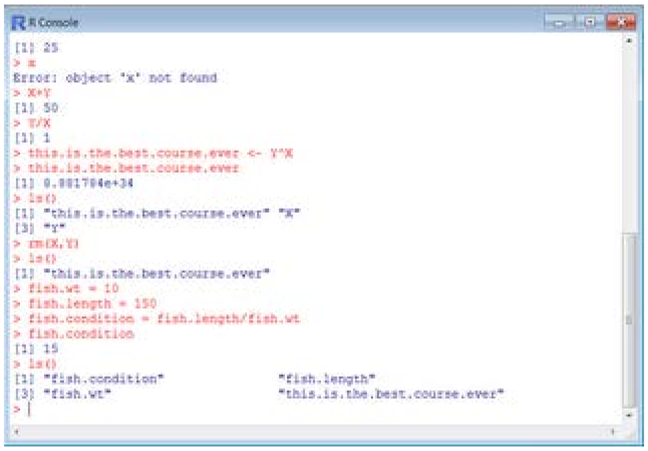
```
fish.wt = 10
fish.length = 150
fish.condition = fish.length/fish.wt
fish.condition
ls()
```

Then select the commands and right click your mouse as:



```
fish.wt = 10
fish.length = 150
fish.condition = fish.length/fish.wt
fish.condition
ls()
```

Then choose "Run line or selection" (you could also hit Ctrl-R) and you should get the following in the console.



```
[1] 25
> x
Error: object 'x' not found
> X+Y
[1] 50
> Y/X
[1] 1
> this.is.the.best.course.ever <- Y^X
> this.is.the.best.course.ever
[1] 8.881784e+34
> ls()
[1] "this.is.the.best.course.ever" "X"
[3] "Y"
> rm(X,Y)
> ls()
[1] "this.is.the.best.course.ever"
> fish.wt = 10
> fish.length = 150
> fish.condition = fish.length/fish.wt
> fish.condition
[1] 15
> ls()
[1] "fish.condition"               "fish.length"
[3] "fish.wt"                      "this.is.the.best.course.ever"
> 
```

Notice that the console displays the commands just as if you entered them individually and prints out the results. Before we go any further, we need to talk annotation. **Annotating an R script** or any program is always good practice. It helps explain what you did and helps you communicate it to others or helps you remember what you were doing at each step. To annotate a program, we use the pound sign "#" at the beginning of each line and R will ignore anything after the # and before the "return". For example,

```
# This program calculates the goofy Peterson condition factor
fish.wt = 10
fish.length = 150
## here's the formula..., pure genius
fish.condition = fish.length/fish.wt
fish.condition
ls()
```

Everything shown in green is ignored by R when you submit the script. You can use more than one # as shown above. Using the comment character "#" also can be used to turn off parts of programs. For example, adding a # before fish.condition:

```
# This program calculates the goofy Peterson and Colvin
condition factor
fish.wt = 10
fish.length = 150
## here's the formula..., pure genius
fish.condition = fish.length/fish.wt
#fish.condition
ls()
```

stops R from printing out fish.condition after it is calculated. Go ahead and try it. As we will find later, this feature is very useful for debugging programs. We heartily recommend using this feature to take notes during the course too. Go ahead and save this script, we may use it later.

As we discussed earlier, the objects we created are written to the working directory. If you save the contents of the working directory (think: "*massive database that I worked on for hours*"), they will always be available when you start R. But where is the working directory? By default, it's more than likely placed somewhere you don't want it to be (but see profile below). Therefore, it's always good practice to **set the working directory** at the beginning of your R session. We do this using the "`setwd(path)`" command, where path is the windows path to the folder where you want to write and read things. For example, let's say that I wanted to save this session to my memory stick and the memory stick was assigned as the "G" drive. I want to save it in a folder "RRRR" inside of "Jims class stuff":

```
setwd("G:/Jims class stuff/RRRR") ## NOTICE USE OF FORWARD SLASH
# This program calculates the goofy Peterson and Colvin condition factor
fish.wt = 10
fish.length = 150
## here's the formula..., pure genius
fish.condition = fish.length/fish.wt
fish.condition
ls()
```

Or you could use the double back slash. *R will not recognize single backslash format.*

```
setwd("G:\\Jims class stuff\\RRRR") ## NOTICE USE OF DOUBLE BACKSLASH
# This program calculates the goofy Peterson and Colvin condition factor
fish.wt = 10
fish.length = 150
## here's the formula..., pure genius
fish.condition = fish.length/fish.wt
fish.condition
ls()
```

To save the all of the contents of your working directory you ca use "`save.image(file = "myfilename.Rdata")`" command. This command saves everything to "myfilename.Rdata" where myfilename would be your file name. Alternatively, you can save specific objects in the working directory to a file using "`save(file = "myfilename.Rdata",list = c(`*`names of objects in quotes separated by commas`*`))`". For example, let say that we wanted to save everything we created using the above script to a file called "firstRclass.Rdata":

```
setwd("G:\\Jims class stuff\\RRRR")
# This program calculates the goofy Peterson and Colvin
condition factor
fish.wt = 10
fish.length = 150
## here's the formula..., pure genius
fish.condition = fish.length/fish.wt
fish.condition
ls()
### Here's where we save everything
save.image(file = "firstRclass.Rdata")
```

This would save all of the objects to *firstRclass* in the folder specified with the `setwd` command above. Alternatively, let's say that we wanted to only save *fish.wt* and *fish.condition*. We would specify:

```
### Here's where we save just 2 objects
save(file = "firstRclass.Rdata", list =
c("fish.condition","fish.wt"))
```

Whew, we saved all of that work, turned off the computer, and headed to happy hour. How do we access the objects that we saved? No problem, we use the "`load`" command, but first we *have to set the working directory*.

```
   load("firstRclass.Rdata")
```

You also may encounter another method of specifying the current working directory in the file path using "`./`" before the filename. For example:

```
    load("./firstRclass.Rdata")
```

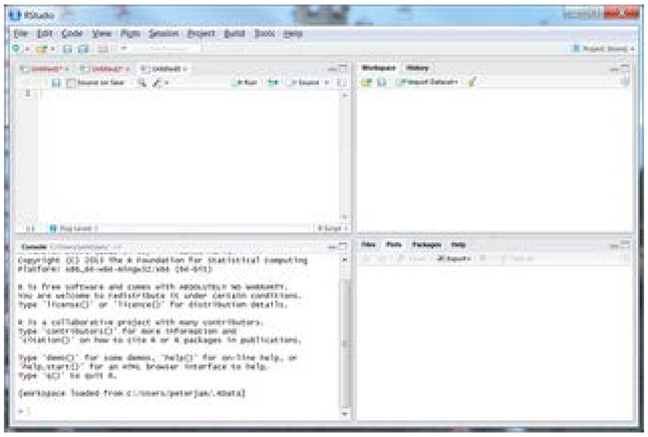
In the standard R GUI and Rstudio (more below), we won't have to use "`./`" but it can come in handy later.

Another way we could have loaded the file was to specify the path inside the `load` command.

```
# Load objects from existing file, full path edition
load("G:\\Jims class stuff\\RRRR\\firstRclass.Rdata")
```

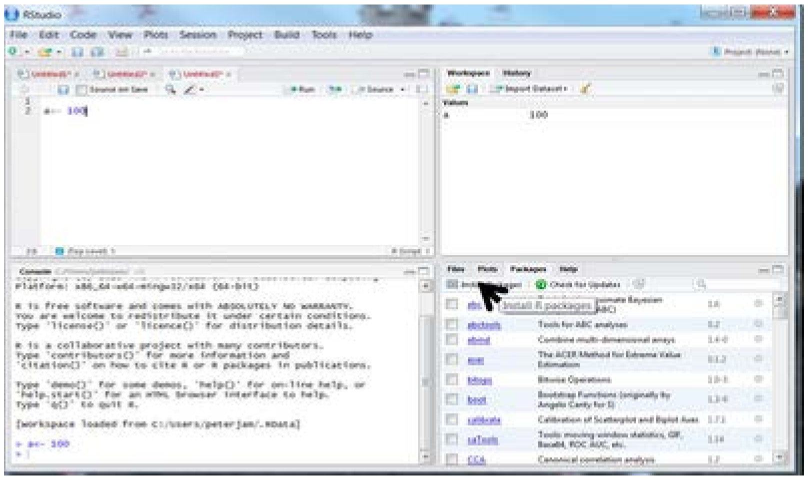
Rstudio

Hey-- what about Rstudio? We haven't forgot and best of all, you haven't wasted your time. Almost everything we just learned is applicable to Rstudio. You still use scripts, the same commands, and have to load packages. The benefits of Rstudio is that the added additional windows like functionality. R studio is just a useful GUI, so R must be installed before installing Rstudio. Once you install Rstudio, open it and it should look something like this:


RStudio
File Edit Code View Plots Session Project Build Tools Help
Untitled1 Untitled2 Untitled3
Source on Save
Run
Source
Workspace History
Import Dataset
Console
Files Plots Packages Help
Export

```
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from c:/Users/peterjan/.RData]

>
```

Rstudio allows you to customize your environment and use multiple panes. Here I have 4 panes shown. Upper left is an Untitled R script; upper right will display all R objects created or read in the R session (workspace tab) or all of the commands used (History tab); the lower left is the console (just like the base R GUI); and the lower right can display the installed packages, help files, plots, and all files in the working directory. These windows can be moved around or closed. The beauty of Rstudio is that it will automatically load packages for you (more later).



```
a<- 100
```

```
a          100
```

```
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from c:/Users/peterjan/.RData]

> a<- 100
>
```

abc	[illegible] Approximate Bayesian [illegible] (ABC)	1.6	
abctools	Tools for ABC analyses	1.2	
abind	Combine multi-dimensional arrays	1.4-0	
acer	The ACER Method for Extreme Value Estimation	0.1.2	
bitops	Bitwise Operations	1.0-5	
boot	Bootstrap Functions (originally by Angelo Canty for S)	1.3-4	
calibrate	Calibration of Scatterplot and Biplot Axes	1.7.1	
caTools	Tools: moving window statistics, GIF, Base64, ROC AUC, etc.	1.14	
CCA	Canonical correlation analysis	1.2	

Highlight the code in the script pane and click "Run" in the upper right hand corner of the script pane. The console should show the commands and the output from the commands just like it did in base R GUI. You also should be able to see the names of the dataframes that you created in the Workspace pane, like this:

Double click on one of the files and it will open up in a separate tab in the script pane. You can now view objects without having to print them to the command console. There are many, many other things Rstudio can do in a windows-like environment, such as reading text files, saving objects or a session to a file, setting a working directory, and several others. Mess around with the GUI and we're sure you can find others. One more thing about the script editor. You should notice that the script editor automatically colors comments and things inside of quotes green. It also colors commands blue and will help you debug problems. This increased functionality can really be helpful when learning R.

CREATING AN R OBJECT

We learned to create a simple object consisting on a single number using one of two assignment operators “=” and “<-". For example, if we enter the code below we create an object "Z" and assign it a value of 10.

```
# using the = operator
Z= 10
# or using the assignment “<-“ operator
Z <- 10
```

Vectors

These single value objects are often referred to as scalars if they represent numeric values (i.e., a single number). To create an object consisting of several numbers (often called a vector), you need to use the collection operator “`c(numbers in here separated by a comma)`”. For example, let’s create a vector named “jims.vect” consisting of the numbers 1 through 5.

```
# create vector
jims.vect <- c(1,2,3,4,5)
# print out vector
jims.vect
```

As we will learn later, we can perform operations on entire vectors and individual elements or groups of elements of vectors. For example, we can create another vector that contains the values of jims.vect divided by ten using the following:

```
# divide all the values in the vector by 10
new.vect <- jims.vect/10
# print out vector
new.vect
```

Vectors consist of elements that are indexed by the order in which that occur. For example the third element of "new.vect" is 0.3. We can refer to the elements of a vector using a bracket "[]" with the number in the bracket referring to the element in the vector. For example, let's print out the 4th element of *new.vect*:

```
# print out 4th element
new.vect[4]

## assign 4th element to another object, fourth
fourth = new.vect[4]
```

We can also refer to multiple elements in a vector. For example:

```
# print elements 3, 4, and 5 in new.vect
new.vect[3:5]
```

Notice above that we used a colon (:) to refer to a sequence of elements from 3 to 5. This sequence notation will come in handy and has other uses. For example, let create a vector “biggy” consisting of integers (whole numbers) from 10 to 20:

```
## create vector
biggy = c(10:20)
#print out vector
biggy
```

Notice that it uses the collection operator “c()” and the list separator “:”. There are all kinds of neat functions in R for creating vectors. Below are a few that we find useful with a description of each in the comment above.

```
# create a vector with a sequence of values from 1.25 to 8.75
by increments of 0.25
wow <- seq(1.25, 8.75, by = 0.25)

# create a vector with a sequence of 13 evenly spaced values
between 9 and 14
double.wow <- seq(9, 14, length = 13)

# create a vector of 13 elements with the same value 41
double.dog.wow <- rep(41,13)

# create a vector consisting of two sequences of 1,2,3,4
triple.dog.wow <- rep(1:4, 2)
```

Be sure to try each of these methods and print out the results. There are other uses for these functions so be sure to use “help()” to see what is possible.

You may be wondering, what if I want to select non-consecutive elements of a vector? No problemo, we have a list of the value inside a the collection operator that is inside of the brackets. You’re probably asking: *What the @#$%% are you guys talking about?* Here-- let’s show you below. First, let’s create a vector consisting of values from 1 to 10 by increments of 0.5, then select the odd numbered elements.

```
# create a vector with a sequence of values from 1 to 10 by
increments of 0.5
nuts <- seq(1,10, by = 0.5)

# select the odd numbered elements of nuts and put them into a
vector wing.nuts
wing.nuts = nuts[c(1,3,4,5,7,9,11,13,15,17,19)]
```

Typing in all of those numbers in the list to get wing.nuts was a pain. Notice that the numbers consisted of integers that went from 1 to 19 by increments of two. Above, we learned that we could generate a sequence of numbers using “seq”. Hmm… wonder if we can combine these two ideas…let’s try:

```
# select the odd numbered elements of nuts using seq and put
them into a vector wing.nuts
wing.nuts = nuts[seq(1,19,2)]
```

Wow that worked! There are all kinds of tricks like this that can be used in R and the only way to discover them is to try different things. Use your imagination and don’t be afraid, try something else. Seriously--- we're waiting here till you try to combine a couple of commands.The worst that could happen is the dreaded red error message. Trial and error is how most of us learn these tricks.

Matrices

We just learned to create single value (scalars) objects and a vectors consisting of a single row or single column of values. Matrices consist of multiple values contained in multiple rows and columns. For example, the matrix below consists of 4 rows and 3 columns and can be referred to as a 4 by 3 matrix (or 4x3 matrix):

```
 1  2  3
 4  5  6
 7  8  9
10 11 12
```

We can create this matrix in R using the "`matrix`" function (*who'd a thunk it?*). First, let's create a vector with values from 1:12.

```
### create the vector the hard way
vect = c(1,2,3,4,5,6,7,8,9,10,11,12)

### create the vector the easy way
vect = c(1:12)
```

Now we can use the `matrix` function, but note that we need to specify the number of rows or columns using the "`nrow`" ***or*** "`ncol`" options, respectively. We may need to specify one more thing. Let's see what happened without it.

```
### create the vector the easy way
vect = c(1:12)
## create the 4 by 3 matrix using the values in vect
jims.matrix = matrix(vect, nrow = 4)
## create the 4 by 3 matrix using the values in vect
jims.matrix = matrix(vect, ncol = 3)
```

The matrix `jims.matrix` should look like this when printed:

```
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Notice that the values are in order going down the first column 1-4. This is not quite what we have above where the values are in order across rows. This is because the default for the matrix function is to place values from the matrix by column. This mean that the first 4 elements on "vect" are the values in the first column, the second 4 values in "vect" are in the second column and so forth. We can get a matrix like the original one above specifying "`byrow = TRUE`" option in the matrix function as:

```
## create the 4 by 3 matrix using the values in vect
jims.matrix = matrix(vect, ncol = 3, byrow = TRUE)
## print out the matrix
jims.matrix
```

When you execute the above code you should get:

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
```

This looks just like the above (original) matrix. Similar to vectors, we can refer to elements in a matrix using numbers inside if brackets corresponding to the row and column contain the element separated by a comma: `matrix.name[row,column]`. The first number always refer to the row copy and paste this code in R and see what happens:

```
### print out value in row 2, column 3 in jims.matrix
jims.matrix[2,3]

## print out the values in rows 2 and 3 in the first column
jims.matrix[2:3,1]

## print out the values in rows 1 and 3 in the second column
jims.matrix[c(1,3),2]

## print out all the rows for columns 1 and 3, notice blank
for row
jims.matrix[,c(1,3)]
## print out all the columns for row 2, notice blank for
column
jims.matrix[2,]
```

You should notice the use of the sequence operator “`:`” and collection operator “`c()`” to refer to specific rows and columns. You should also notice that you refer to all rows or all columns by leaving the value for the row or column blank. We can also change the values of specific elements of a matrix (or vector) using the above notation. For example,

```
## change the value in row 2, column 1 to -99
jims.matrix[2,1] <- -99
## print out matrix
jims.matrix
```

```
## change the values in rows 1 and 4, column 3 to missing,
remember NA is missing
jims.matrix[c(1,4),3] <- NA
## print out matrix
jims.matrix

## change the values in row 4, column 2 to the sum of values
in row 3, columns 1 and 2
jims.matrix[4,2] <- jims.matrix[3,1] + jims.matrix[3,2]
## print out matrix
jims.matrix
```

Go ahead and try different things, but be forewarned that any operation on a missing value (NA) will result in NA.

Working with character values and strings

So far, we have only assigned numeric values and missing values to R objects. Many times we may wish to assign characters (e.g., a, b, c) or a string of characters (dog, cat) to an R object. For example, we may want a species or site name. To distinguish numeric and character variables, we place the latter (character) inside ***single quotes.*** For example, let’s create an R object species and assign it a value 'dog'

```
## create species assign “dog”
species = ‘dog’
#print out
species
```

Similar to numeric values, we also can create a vector that contains several strings or character values using the collection operator. Note that we must ***use double quotes in place of single quotes*** to delineate the character strings.

```
## create species.vect and assign pet names
species.vect = c("dog","cat","hamster")
#print out
species.vect
```

(WARNING, WARNING: copying and pasting the above may result in errors because R or Rstudio may not recognize the quotes as quotes, so if an error occurs. Type in the code by hand)

There also are neat tricks for working with characters, similar to tricks with numeric values. For example, let's say I want to create a vector containing the letters a to g in order.

```
# create vector alphabet the hard way
alphabet <- c("a","b","c","d","e","f","g")
#print it out
alphabet

# create vector alphabet the easy way
alphabet <- letters[1:10]
#print it out
alphabet
```

We also can create a matrix that contains several characters/strings using the matrix function.

```
## create pet.vect and assign pet names
pet.vect = c("dog","cat","hamster","goldfish","mouse","bird")
## create a matrix
pet.matrix = matrix(pet.vect, ncol = 3, byrow = TRUE)
## print it out
pet.matrix
```

(WARNING, WARNING: copying and pasting the above may result in errors because R or Rstudio may not recognize the quotes as quotes, so if an error occurs. Type in the code by hand)

Pretty neat, eh? We can refer to specific elements in character/string vectors and arrays using the exact same notation that we used with numeric values. For example,

```
## print out the values in row 1 column 1 and 3
pet.matrix[1,c(1,3)]

## print out all the rows for columns 1
pet.matrix[,1]
```

Try some other combinations of rows and columns just to get the hang of it.

The preceding may have spawned an idea, maybe I can create a matrix with numeric values and characters. Let’s try that Ok, we want to create something that looks like this matrix:

```
a    1    5
b    2    6
c    3    7
d    4    8
```

To do this, first create a vector that contains all of these elements

```
# the hard way
trial<- c("a","b","c","d",1,2,3,4,5,6,7,8)
#print it out
trial

#easier way
trial<- c(letters[1:4],1:8)
#print it out
trial
```

Did you notice anything interesting when you printed out the vector? Maybe you noticed the fact that the numbers that printed out had double quotes around them like this:

```
[1] "a" "b" "c" "d" "1" "2" "3" "4" "5" "6" "7" "8"
```

This means that R *is treating numeric variables just like characters*. If fact, a vector or a matrix cannot contain mixtures of numeric and character variables. Just to emphasize:

A vector or a matrix cannot contain mixtures of numeric and character variables in R.

R automatically treats all of the elements as character variables. What does that mean???? Before we investigate that, let's finish what we started and create the matrix with the "trial" vector. We will use the matrix function to create the matrix. We can see above that we want 3 columns so "`ncol = 3`" we also see that we want the first 4 elements of "trial" to be the first column of the matrix, so we want the function to place elements in the matrix by column and "`byrow = FALSE`":

```
## create mixed up matrix
trial.n.error <-matrix(trial,ncol = 3, byrow = FALSE)
# print it out
trial.n.error
```

and you should obtain

```
     [,1] [,2] [,3]
[1,] "a"  "1"  "5"
[2,] "b"  "2"  "6"
[3,] "c"  "3"  "7"
[4,] "d"  "4"  "8"
```

Just like the before, R put double quotes around the numbers. That is, it considers the numbers in columns 2 and 3 to be characters. Ok, now we are ready to address: *what does this mean????* We learned above that we can perform operations on specific elements of a vector and matrix.

Let’s see what happened when we attempt to add columns 2 and 3:

```
## add columns 2 and 3 on trial.n.error matrix
trial.n.error[,2] + trial.n.error[,3]
```

after submitting this code to R you should have gotten

```
Error in trial.n.error[, 2] + trial.n.error[, 3] :
  non-numeric argument to binary operator
```

This is because R thinks the numbers are letters. We can determine the type and properties of objects using several R functions. Among the most useful functions is `typeof`. Go ahead and find out what the function does using help, i.e., `help(typeof)`. You should see its syntax and various uses. Let find out what type object `trial.n.error` is.

```
  ### what is this
  typeof(trial.n.error)

  ##we can do the same for individual columns or rows
  typeof(trial.n.error[,2])

  ## what class of object is trial.n.error
  class(trial.n.error)
```

As you should see, the printout in the console indicates that the objects is “character” and the class function tells us that it is a matrix. Try using the `typeof` and class functions on some of the other objects. For Rstudio users, you should be able to see that the characteristics or each object in the workspace in the “workspace” window.

We can use other functions, in particular the "`is.`" functions, to determine the type of object here are two "`is.`" functions below:

```
## is trial.n.error a numeric matrix
is.numeric(trial.n.error)

## is trial.n.error a character matrix
is.character(trial.n.error)
```

We can change the characteristics of an object (this is caller *coercion* of an object) using the "`as.`" functions. To illustrate, create an object "fix" that contains columns 2 and 3 from "trial.n.error". Then let's check out the characteristics of the object.

```
# create new object
fix<- trial.n.error[,2:3]

# what is this type of object
class(fix)

# is fix a matrix, notice another is function-- of course!
is.matrix(fix)

## what are the type of variables in fix
typeof(fix)

## what are the attributes of fix, yes another function
attributes(fix)
```

The R console should have indicated that fix was a matrix, (is.matrix = TRUE), containing character values, and the dimensions of the matrix, `$dim`, is 4 by 2. Notice that the characteristics of the fix matrix are the same as the `trial.n.error` matrix, with the exception of a missing column. This means that the new objected inherited the characteristics of its parent object. This idea of inheritance is a very important concept. Any object that you create from another object will have the SAME characteristics of the first object. *Allllrighty*, let's change fix to become a numeric matrix:

```
# coerce fix to become numeric
fixed <- as.numeric(fix)
#print it out
fixed

# what is this type of object-- numeric
class(fixed)

# is fixed a matrix, -- oh no!
is.matrix(fixed)

## what are the type of variables in fixed-- yes numeric!
is.numeric(fixed)
```

What happened? Fixed contain numeric values but is no longer a matrix. It is now a vector. How do we get a matrix back?... maybe the `matrix` function?:

```
fixed <- as.numeric(fix)
#print it out
fixed <- matrix(fixed,ncol = 2, byrow = FALSE)
fixed
```

```
# what is this type of object-- matrix
class(fixed)
```

Ok, that worked. You also could have created a couple of different ways. Can you think of any? Go ahead any try one or two. We can now add the two columns. In fact we can create a new object as the sum of columns 1 and 2

```
#add column 1 and 2 of fixed
fixed[,1] + fixed[,2]

#create a third column in fixed by adding column 1 and 2 of
fixed
new.val <- fixed[,1] + fixed[,2]

#print it out
new.val
```

Hey, what happens if we coerce letters to make them numeric values?

Well, let's try it and find out using the character matrix trial.n.error:

```
# create vector curious by coercing trial.n.error
curious<-as.numeric(trial.n.error)
## print it out
curious
```

I created numeric values in a vector, but it also assigned the missing values (NA) to the instances where it tried to coerce an actual character. How do we find out is an object contains missing values? Hmmmm… if we want to find out if an object is a character we use "in.character", if we want to find out is an object is numeric we use "is.numeric", what if we want for found out is something is NA (missing). If you’re thinking “is.na” then you are correct. Let’s try that:

```
## are there missing values in curious
is.na(curious)
```

You should get:

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

Notice that the value is `TRUE` when the element in curious is `NA` and `FALSE` when it isn't.

The “`is.na`” function is very, very useful and will come in handy. To illustrate, let’s say that you wanted to replace the missing values with zero or some other default value, say `-666`, you would use the following:

```
#replace missing elements in curious with zero
curious[is.na(curious)] <- 0
#print it out
curious
```

Notice the syntax, R replaces the elements in the vector where `is.na(curious)` is TRUE. Hmmm maybe this means that any element that meets some criteria can be changed? Let's try. How about changing all zeros to -666 in other words change all the values where `curious == 0` is TRUE.

```
#replace zero elements in curious with -666
curious[curious== 0] <- -666
#print it out
curious
```

Finally in this subsection we will address a question in the back of your mind. Can we coerce a numeric variable to become a character variable, why yes! And you can probably guess what function we will be using: "as.character".

```
## create numeric vector num.vect with values 1:15
num.vect = c(1:15)
## print it out
num.vect

# create character vector char.vect by coercing num.vect
char.vect= as.character(num.vect)
## print it out
char.vect
```

Yes, coercion can be a good thing.

Data frames

It may be a bit confusing that matrices cannot contain both character and numeric values. Data frames can contain both numeric and character values. The only caveat is that the variables in a column must be the same type. For example, a column can consist of either all character/string variables OR numeric values NOT BOTH. So how do we create a data frame? One way is to read in an external file. Another way is coercing an existing object into a data frame. If you are thinking "I can use as.data.frame" you deserve a gold star… or maybe a beer. Let's try this with the trial.n.error matrix.

```
## print out matrix as a reminder
trial.n.error
#create my.dater data frame by coercing trial.n.error
my.dater<- as.data.frame(trial.n.error)
# print it out
my.dater

#what classs of object
class(my.dater)

# is my.dater a data frame-- yes!
is.data.frame(my.dater)

# is my.dater a matrix -- no!
is.matrix(my.dater)
```

So now you have a data frame with default column names V1, V2, and V3. We can change the names of the columns to something more meaningful using the colnames function.

```
#change column names in my.dater to pixie, dixie, and bud
colnames(my.dater) = c("pixie", "dixie", "bud")
#print it out
my.dater
```

New trick: We can refer to individual columns in a data frame using the $ syntax for example:

```
    #print out the second column dixie in my.data
    my.dater$dixie

    #we can also refer to individual columns in a data frame
    using brackets
    #print out the second column dixie in my.data
    my.dater[,2]
```

So, you can specify elements in a data frame using column names and using a row and column index inside a bracket [], just like a matrix. You should also have noticed something in the console output.

```
   > my.dater[,2]
   [1] 1 2 3 4
   Levels: 1 2 3 4
```

The “`Levels`” thing at the bottom. *What the heck is that?* It indicates that the R considers the variable type in the `dixie` column to be a factor, which is a class variable used in an analysis. For example, you might use a factor variable in an ANOVA to represent study site when you test for differences among study sites. We discuss more use of factors later, for now it’s important to note that a factor is basically treated like a character variable in that they are not numbers and cannot be treated as numbers. For example, you cannot add them. To convince yourself, add the columns `dixie` and `bud`:

```
#try to add dixie and bud columns in my.dater
my.dater$dixie + my.dater$bud
```

you should get

```
[1] NA NA NA NA
Warning message:
In Ops.factor(my.dater$dixie, my.dater$bud) : + not meaningful
for factors
```

We can determine if a variable is `numeric` or `factor` using the class command or an "`is.`" function.

```
# whatclass is dixie
class(my.dater$dixie)

#is dixie numeric- no!
is.numeric(my.dater$dixie)

#is dixie a factor- yes!
is.factor(my.dater$dixie)
```

Commonly Encountered Problems (CEP): Note that R sometimes changes numeric variables into factors when you read in a dataset. You will be unable to perform numeric operations on these factor variables and you will be quite frustrated! Bottom line: check the data in each column and make sure it is the correct type. (More later)

The only way we can change these to numbers is to coerce them. Let's try first using what you may expect to use "as.numeric"

```
# first print out bud
my.dater$bud

# try to coerce bud to a numeric value
as.numeric(my.dater$bud)
```

Notice that it changed the values 5,6,7,8 to 1,2,3,4. ***This is a bad thing*** ---> so here's is the bottom line:

IMPORTANT POINT: when coercing factors to numeric values-- you must first coerce them to character variables, then coerce the character variables to numeric variables

The correct way to change numbers is to coerce them, so first "as.character" then "as.numeric"

```
# print out bud
my.dater$bud

# now coerce as character then to coerce bud to a numeric
value
as.numeric(as.character(my.dater$bud))
```

Now let’s fix both dixie and bud columns, be sure to check the class.

```
# now coerce as character then to coerce bud to a numeric
value
my.dater$bud <- as.numeric(as.character(my.dater$bud))

# whats the class-- numeric!
class(my.dater$bud)

# is it numeric--yes!
is.numeric(my.dater$bud)

# now coerce as character then to coerce dixie to a numeric
value
my.dater$dixie <- as.numeric(as.character(my.dater$dixie))

# whats the class--numeric!
class(my.dater$dixie)

# is it numeric--yes!
is.numeric(my.dater$dixie)
```

Wasn’t that bloody good fun?

Lists

Lists are the final types of R objects that you will commonly encounter. They also are the most frustrating and (at least for beginners) difficult to work. Lists can contain objects of various types, numeric, character and lists are the default output that is created when you conduct a statistical analysis, such as linear regression. We will discuss using lists resulting from statistical analysis later in the course. For now, let’s go over some of the basics with lists. To create a list we use the.. (wait for it)... `list` function. Let's create several different types of objects and combine them in a list.

```
# create a numeric vector with a sequence of values from 1.25
to 8.75 by increments of 0.25 and print
num.vct <- seq(1.25, 8.75, by = 0.25)
num.vct

# create 3 by 4 numeric matrix with a sequence of values from
1 to 6.5 by 0.5 and print
num.mtrx <- matrix(seq(1, 6.5, by = 0.5), ncol = 4, byrow =
FALSE)
num.mtrx

# create a character vector with a through z and print
char.vct <- letters[1:10]
char.vct

# create 2 by 2 numeric with peoples names and print
char.mtrx <- matrix(c("bill", "mary","joe","brenda"), ncol =
2, byrow = FALSE)
char.mtrx

## create a list that contains all of these objectives
big.list <- list(num.vct,char.mtrx,num.mtrx,char.vct)

## create a name for each object within the big.list and print
names(big.list) <- c("vect_numbrs", "names",
"numb_matrx","letters")
big.list
```

Let’s find out what type of object we just created.

```
## what class is this object
class(big.list)

# what type of object
typeof(big.list)

## new very important function for lists
str(big.list)
```

The last of these functions `str()` is particularly important and provides information on the objects within the list and the names of the objects, so let’s examine the output:

```
List of 4
 $ vect_numbrs: num [1:31] 1.25 1.5 1.75 2 2.25 2.5 2.75 3
3.25 3.5 ...
 $ names      : chr [1:2, 1:2] "bill" "mary" "joe" "brenda"
 $ numb_matrx : num [1:3, 1:4] 1 1.5 2 2.5 3 3.5 4 4.5 5 5.5
...
 $ letters    : chr [1:10] "a" "b" "c" "d" ...
```

The above was output to the console resulting from `str(big.list)`. It tells us that big.list contains 4 objects (`List of 4`), and each object has a name (names are after $). The first (`$ vect_numbrs`) consists of numbers in a vector with 31 elements (`num [1:31]`), the second (`$ names`) consists of a 2 by 2 matrix of characters variables (`chr [1:2, 1:2]`), the third object (`$ numb_matrx`) is a 3 by 4 matrix of numeric variables (`num [1:3, 1:4]`), and the 4th object (`$ letters`) is a character vector with 10 elements (`chr [1:10]`).

We can access the elements of a list using syntax similar to that is used to access elements within a data frame, using $ and brackets [].

```
# to access the ‘names’ object in big.list
big.list$names

# to access the names object the 2nd one in big.list
big.list[2]
```

Hmm looks too easy, right? Better check on the characteristics or each of these objects.

```
# what is the class of object created using $ in big.list
class(big.list$names)

# what is the class of object created using [] syntax in
big.list
class(big.list[2])
```

Woa! The first method ($) produces a matrix and the second a list. This is important because you can’t really do much with a list in terms of common operations (e.g., addition, multiplication, etc.) but you can with matrices and vectors. To illustrate, try multiplying the first object in the list, vect_numbrs, by 5.

```
#multiply the elements in the first object within the list by
5
big.list$vect_numbrs*5

#multiply the elements in the first object within the list by
```

```
5
big.list[1]*5
```

You should have noticed that the first methods resulted in products, whereas the second method produces the error message:

```
Error in big.list[1] * 5 : non-numeric argument to binary
operator
```

The importance of accessing objects in a list will be very apparent when we start using the output from statistical analysis, such as plotting residuals.

READING AND WRITING TEXT DATA FILES

Reading files

Specifying paths inside commands can be used for several other commands, but we think it is better a practice when learning R to set the working directory and read and write files from that directory. Speaking of reading and writing files, there are several commands that can be used to read and write files in a variety of formats. For now, we'll stick to datafiles. The two most common formats read (and written) are text files in tab or comma delimited formats. Delimited means how the data in columns are separated. In a tab delimited file, the data in each column in a row are separated by a tab. Here is a simple tab delimited file with a header (the first row contains the column names):

```
pet     length     wt     age
cat     100     25     15
dog     500     257     5
```

and the same file in comma delimited format.

```
pet,length,wt,age
cat,100,25,15
```

```
dog,500,257,5
```

Save the full version of the pets data in tab (*pets.txt*) and comma (*pets.csv*) delimited versions by right clicking on the links and saving to your computer/ memory stick. These files were created in excel using "save as" and selecting "Text (Tab delimited) (*.txt)" and "CSV comma delimited (*.csv)" options. Go ahead and open them up and take a look. Nothing scary there. As with everything in R, there are several ways to read in data files. We will learn to use two methods for text files: `read.table`, which can read in text files with any type of delimiter and `read.csv`, which reads in comma delimited files. Remember to set your working directory to the location of the text files. Here's the syntax for reading in the tab delimited pets data:

```
# set working directory
setwd("G:/Jims class stuff/RRRR")
## read in tab delimited file
pet.data<-read.table("pets.txt", header = TRUE, sep = "\t")
### print the contents to the console
pet.data
```

Here the name of the file is specified first. We then indicate that the file contains a header (column names), `header = TRUE;` and we specify that the file is tab delimited using `sep = "\t"`. Whew, how can anyone remember all that? We can't. That's why we always check the syntax using the help function. Let's do this by typing `help(read.table)` in the console. You can see all of the options. For example, if the file did not contain column names we would specify `header = FALSE` (note upper case of FALSE). If you review the help file, you'll see that you can specify the type of delimiter with the `sep` option. Hmmmm... `"\t"` represented tab delimited... wonder what we use for comma delimited? Maybe a comma `","`. Let's try that.

```
## read in comma delimited file
```

```
pet.data2<-read.table("pets.csv", header = TRUE, sep = ",")
### print the contents to the console
pet.data2
```

It worked. Now let's try using `read.csv` to read in the pet data. Let's use the help command to pull up the help file and look at the syntax. Based on the help file our command should look something like this:

```
## read in comma delimited file
pet.data3<-read.csv("pets.csv")
### print the contents to the console
pet.data3
```

Notice how many fewer options we needed to specify. This was because the default was header = TRUE. **Important point to remember: the default options for any R command/function are displayed inside the command at the top of the help file.**

Commonly Encountered Problems (CEP) reading data:

Save the comma separated file *Habitat data.csv* to your computer and open using Excel. These data were collected in wetlands in central Utah. There are 12 columns of variables ranging from site number to average water depth (Avg H20) and there are 29 observations (rows). Read the data into R using `read.csv` command-- *be sure that the working directory is correct*-- and print the contents e.g.,

```
habitat<-read.csv("habitat data.csv")
habitat
```

What happened? Does it look like the same data in excel? It shouldn't. You should see several changes.

1) There are a whole bunch on "`NA`". In R, "`NA`" means that the data are missing.
2) The column names have changed. For example, we now have `"Site.."` rather than "`Site #`"
3) The print out also indicates that there are columns `X, X.1, X.2...X.10`.
4) There are also 35 rows of data (all `NA`) rather than 29.

What the... @#$#@&!!? This happens all of the time (even to us).

1) The NA are not necessarily a problem. The original data did have some missing data, so R simply replaces missing data with NA. We will learn how to deal with real missing data later in the course.
2) The column names changes because R does not allow spaces or special characters ("#", "(",")") in the names of objects and columns and it will automatically change these to periods ".". SOLUTION: Don't use special characters.

3-4) The extra columns and rows are because the csv file contains extra rows and columns. This is often due to a stray character or space in the file. For example, take a look at Habitat data.csv in Excel and you will see a stray character (x) in column W row 36.

SOLUTION: Delete the extra rows and columns.

You probably noticed in the above CEP that it was a pain to print out a moderate size dataset in the console. Using the standard R GUI, you may not want to print out the contents of a gigantonormous file. You can use the command `head()` to print out the column headings and the first 6 lines, e.g.,

```
habitat<-read.csv("habitat data.csv")
## print out first 6 lines
head(habitat)
```

You can also obtain the names of the column headings using the `names` command. For example,

```
habitat<-read.csv("habitat data.csv")
## print out first 6 lines
head(habitat)
## print out column names
names(habitat)
```

We created objects when we read in the text files ans these objects are known as dataframes. They will become among the most important and frustrating objects we use in R. We have entire lessons devoted to working with dataframes. For now, we'll do a couple of operations with the dataframes. To access or use the columns of the dataframe, we need to us a special syntax that involves a dollar sign "$". To refer to the `length` and `wt` columns in pet.data, we use `pet.data$length` and `pet.data$wt`. We can create an R object that contains the data from a column of the dataframe, e.g.,

```
weight<- pet.data$wt
leng <- pet.data$length
```

We also can create a new variable (column) in `pet.data` using the "$". For example, let's say that we wanted to create a variable `leng.wt` by dividing the weight of each per by its length. Simple, we do the following:

```
pet.data$leng.wt <- pet.data$wt / pet.data$length
```

or we also could have done

```
weight<- pet.data$wt
```

```
leng <- pet.data$length
pet.data$leng.wt <- weight / leng
```

If time allows, go ahead and create a couple more variables for the `pet.data` dataframe.

Writing dataframes to text files

Ok, we learned to read files and conduct simple manipulations with dataframes. Now we'll learn how to write data to a text file. The people that gave us R are fairly logical people. The created `read.table` and `read.csv` to read text files. What are the commands for writing files? If you guessed `write.table` and `write.csv` you deserve a gold star. How do we figure out the syntax? All together now..."USE THE HELP COMMAND." Let's do that. We see that just like read.table, write.table can write text files with any type of delimiter, so to write pet.data to a tab delimited file we can use:

```
## Write a tab delimited file to working directory
write.table(pet.data, "Look and me.txt", sep = "\t")
```

This will write the file to your working directory so make sure it is correct!!! Notice that the syntax is similar to read.table. We can do the same and write a comma delimited file using `write.table` and `write.csv`, e.g.,

```
## Write a comma delimited file to working directory
write.table(pet.data, "Look and me.csv", sep = ",")
## Write a comma delimited file to working directory
write.csv(pet.data, "Look and me too.csv")
```

We can read files we can write files, isn't this awesome?

SELECTING AND SUBSETTING DATA, APPENDING, MERGING, SORTING AND DUPLICATING

Selecting and subsetting what is the difference?

Well they are related. In R you can select data and view it manipulate it, and so on. Subsetting takes selecting a step further and makes a new object. Remember that R is an object oriented language.   So you can select parts of an object and you can subset objects to make a new object. We will start out by selecting parts of an object and this will lead into subsetting.

Selecting

What can we select when it comes to R objects?   Let's give it a shot. First we will generate some data to play with. The code below makes 3 vectors, 2 dataframes, and 1 matrix.

```
# lets make 3 vectors
lengths<- runif(100,30,500)
wghts<-  0.0002*lengths^3
relative_weight<- wghts/300

# lets make 2 data.frames
data1<- data.frame(year=rep(2009:2012,25), len=lengths,
wghts=wghts, rel_weight=relative_weight)

data2<- data.frame(year=sample(2009:2012, 50, replace=TRUE),
len=runif(50,100,800))
data2$wghts<- 0.0002*data2$len^3.01
data2$rel_weight<- data2$wghts/500

# what are the field names we can select?
names(data1)
```

```
# select the field length
data1$len # returns the data in the length field
data1$wghts # returns the date from the weight field

# lets make a matrix of random numbers
x<- matrix(runif(120,0,1),nrow=10, ncol=12,byrow=TRUE)
```

What about lists? Aren't they kind of like having multiple objects (vectors, dataframes, matrices) in a single object? Yep...lets make one that combines the 3 vectors, 2 dataframes, and 1 matrix from above in a list! Exciting...

```
mylist<- list(lengths=lengths, weights=wghts,
relative_weight=relative_weight,
data1=data1, data2=data2, matrix1=x) # now we have a list!
```

OK great, now how do I select that data from this infernal list? Columns from objects within a list can be selected using the $ notation similar to what we used above. Let's give it a shot---

```
mylist$length # the vector from above
cbind(mylist$lengths,lengths)# they are the same!

mylist$weights # the vector from above
cbind(mylist$weights,wghts)# they are the same!

mylist$data1$year # the vector from above
data1$year
cbind(mylist$data1$year,data1$year)# they are the same!

mylist$data2$len
data2$len
```

```
cbind(mylist$data2$len, data2$len)# they are the same!
```

Pretty easy, right?

Bracket notation

What about bracket notation? I can guaranty that if you stick with R you will use brackets at some point, and probably be glad you did. As we discussed above, brackets are a way to select rows or columns of a dataframe or list. We will start with a dataframe. The first thing to know is how brackets work. Brackets are really nothing to be afraid of as we learned earlier, they index rows first then columns (i.e., dataframe[row,column]). Ok, this is a bit redundant with what we covered earlier but it's important, so let's compare this with our $ notation from above.

Data frames

What if we wanted to select the length field just like we did above? The following bit of code does the same as above for length but with brackets!

```
# SELECT THE 3RD FIELD (LENGTH) FROM THE DATA
data2[ , 3] # THIS RETURNS THE SAME AS dat$len
names(data2) # note that length is the 3rd field
data2$len
data2[ , 4]
data2$rel_weight
```

Where bracket notation gets convenient is when you want to select a couple of fields

```
data2[ ,c(2,3)] # SELECT LENGTH AND WEIGHT FIELDS
data2[ ,c(2,3,4)] # SELECT LENGTH, WEIGHT, AND RELATIVE WEIGHT
data2[ ,c(2:4)] # this is the same as the above line of code
data2[ ,c(1,3:4)] # pretty flexible
```

```
data2[c(1,4,7,19),c(1,3,4)]
```

In the above bit of code there is nothing in the location for rows. This tells R to just get all the rows. In this case, there are 30 rows in the dataframe.

```
data2[c(1:30),3] # returns the same as dat[ ,3]
data2[ , 3] # same as above
```

Just as we did with the column indexes we can select rows that we want. For example, let’s select the first 10 rows of the length and field columns.

```
data2[c(1:10),c(3,4)]
```

we can also select arbitrary rows

```
dat[c(1,4,7,19),c(1,3,4)]
```

Fun huh? You are likely curious how bracket notation might make your life easier? What if you needed to get the mean length for the rows 1, 3, 5, 7, 9, and 11.

```
sum(data2[c(1,3,5,7,9,11),3])
```

Lists

You can also use bracket notation with lists, however we won't cover these in too much detail in this lesson in this lesson. In my experience lists are rarely used in typical day to day R programming but you may like to use them as you get more proficient or if you are extracting elements from a list.

```
# Lists
mylist$data1$log_transformed_len<- log(mylist$data1$len)
mylist$l_weight<- log(mylist$data1$wght) # for weight
mylist$data1$l_weight
```

```
names(mylist)
names(mylist$data1)
```

Now we can get the various bits of data within the list using either $ or bracket notation. Let’s get the means we just calculated for log length and log weight.

```
mylist$data1$l_weight<- log(mylist$data1$wght) # for weight
names(mylist$data1)

# adding some more to the list
mylist$mean_l_weight<- mean(mylist$data1$l_weight)
mylist$mean_l_weight
```

We used bracket notation before with dataframes, can we use them with list? You betcha... Let’s get the first dataframe in the list, this is done using a special 'double' brackets

```
mylist[[1]]# get the first object
mylist[[5]]# get the 5th object
```

Now to index rows and columns within the list element all we need to do is use brackes just as we did for dataframe but with the special double brackets to tell us what object we want.

```
mylist[[5]][1,]
mylist[[5]][,2]
```

Vectors

Think of vectors as a single column from a dataframe. For example we could extract a column of data from our dataframe dat and assign it to object a.

```
victor<- data1$len
victor # the vector
```

We can select things from the vector. Let's select the 3rd value

```
victor[3]
```

How about selecting all the values greater than 300? We can even use the brackets in a function for this too:

```
mean(victor[victor>300])# not the same as
mean(victor)
new_victor<- victor[victor>300]
mean(new_victor)
```

Some useful tricks application of operators (note operator table presented above)

```
dat<- mylist$data2
# LETS ADD A FEW VALUES
dat[rep(c(TRUE,FALSE),25),2]<- -99
dat[sort(rep(c(TRUE,FALSE),25)),3]<- -99
```

This next line of code selects all the cells in the dataframe that equal -99 and assigns them as NA.

```
dat[dat==-99]<-NA
```

Well that was super easy to take care of those pesky missing values. This also brings up another way bracket notation can be useful—dealing with NA values.

Dealing with NA values.—Most data will have missing values, denoted as NA in R. Probably one of the most infuriating things to do in R—besides deal with dates—is dealing with NA values. NA values are identified using the is.na() command. This command returns a TRUE/FALSE if a cell is NA. Using this command we can select the NA values. So if a cell is NA then the function returns TRUE.

```
is.na(dat$len)
```

we can use this in bracket notation as well to select the rows where length data are missing

```
dat[is.na(data$len),]
```

We can also use a conditional expression to select the rows where there length data is not NA.

```
dat_no_na<- dat[!(is.na(dat$len),] # super intuitive right?
dat # not the same as dat_no_na
```

What if we want to just drop the missing values in a vector or dataframe?

```
dat$len
na.omit(dat$len)
```

We can also use this to drop the missing value in a dataframe using vector notation

```
dat[na.omit(dat$length),]
```

A "!" is use for 'not' (see table above).   A similarly useful function is complete.cases(). This function returns a True/False if all the cells in a row (i.e., record) contains values.

```
complete.cases(dat) # returns a vector of true/falses
dat_complete<- dat[complete.cases(dat),] # select rows with
no NA values
dat_complete # look at new data without NAs
dat # the data with NAs
```

Subsetting

Subsetting is essentially the same as selecting, but you think about doing more sophisticated subsets—but you could also do this using bracket notation.

```
dat2010<- subset(data1, year==2010)
dat_after_2010<- subset(data1, year>2010)
dat_all_but_2010<- subset(data1, year != 2010)
dat_even_years<- subset(data1, year %in% c( 2008, 2010, 2012))

Using and in subsetting-R treats and using the "&" sign
dat_2010_length_100<- subset(data1, year==2010 & length > 100)
dat_2010_length_100 # FAIL! X#amp;*^, WHY?
names(data1)
dat_2010_length_100<- subset(data1, year==2010 & len > 100)
```

Using and/or in subsetting-R treats and using the "&" sign and "|"

```
new_dat<- subset(data1, len > 400 | len < 50 )
new_dat<- subset(data1, year == 2010 & (len > 100 | len < 60))
```

Merging

Merging is taking two dataframes and combining them based on a common index or indexes (kinda like the relational databases). Let’s make a dataset to merge with data1.

```
yeardata<-data.frame(year=c(2009:2012),
type=c("wet","dry","extra dry", "bone dry"))
yeardata
```

OK done, we have a dataset that has some yearly data we could use in an analysis. What is common between yeardata and data2?

```
names(data1)
names(yeardata)
```

Well they both have year. It makes intuitive sense to merge year data with year data!

```
mergedata<- merge(data1, yeardata, by="year")
mergedata # cool there is our merged datasets
```

But what happens if all the values of one dataset are not in the other? Let's drop the 2009 year from the yeardata we set up before. We can do this using subset!

```
mergedata<- subset(mergedata, year>2009)
mergedata # no 2009 data!
```

Ok, let's see what happens when we merge that datasets again.

```
mergedata<- merge(data1, yeardata, by="year")
mergedata # uncool, there is no data for 2009
unique(mergedata$year)# lets make sure
```

We can tell R to include those records that do not have year data by specifying all.x=TRUE in the merge function.

```
mergedata<- merge(data1, yeardata, by="year",all.x=TRUE)
table(mergedata$year)
head(mergedata,30)# whew that is what we wanted!
```

We can merge based on 2 fields as well

```
# make some data to merge
data<- data.frame(year=sample(2009:2013, 8, replace=TRUE),
month=sample(7:9, 8, replace=TRUE),
weather=sample(letters[1:3],8,replace=TRUE))
# make some data to merge with that data
lots_o_data<- data.frame(year=sample(2009:2013, 80,
replace=TRUE),
month=sample(7:9, 80, replace=TRUE),
length=runif(80, 100,600))
data
lots_o_data

lots_o_lots_o_data<- merge(data, lots_o_data,
by=c("year","month"),all=TRUE)
lots_o_lots_o_data
```

There is a dataset that was merged based on year and month!

Appending

Adding columns to the side: cbind()

This can be done using the cbind command to making a new object (remember we used it above to look at and compare 2 vectors?). We can do more than 2 if we want

```
all_vectors<- cbind(mylist$length, mylist$weight,
mylist$relative_weight)
```

Just be careful you can cbind anything as long as the number of rows are equal!

Adding rows to the bottom: rbind()

Similarly we can append rows to a dataframe or matrix

Dataframes

```
names(data1)
names(data2)
fulldata<-rbind(data1,data2)
dim(data1)
dim(data2)
dim(fulldata)
```

Matrices

```
class(x) # make sure we can use a matrix
bigmatrix<- rbind(matrix1, matrix1)
dim(matrix1)
dim(bigmatrix)
```

Adding rows to the bottom when column names don't match: rbind.fill()

There are times when you would like to append 2 datasets that do not have every column in common. Solution? rbind.fill() package (more later)

Sorting

You can think about sorting as taking values and putting them in ascending or descending order. For example

```
sort(data2$year, decreasing=TRUE)
sort(data2$year, decreasing=FALSE)

year_sorted<- sort(data2$year, decreasing=FALSE)
cbind(year_sorted, data2$year) # not the same!
```

Ordering is similar but returns the element ids in order. Let's look at what happens when we use order on data2$year

```
order(data2$year, decreasing=TRUE)
order(data2$year, decreasing=FALSE)
order(data2$year,data2$len, decreasing=TRUE)
```

Well those most certainly are not years. They are the order of the element numbers. We can use it to make a new object that will index the rows and make a dataset that is ordered or sorted by some value.

```
orderid<-order(data2$year,data2$len, decreasing=TRUE)
```

We can take that vector and use it as a row index

```
data2_sorted<- data2[orderid,]
```

Or we can feed the function directly in.

```
data2_sorted<- data2[order(data2$year),]
data2_sorted
```

We can also order a dataframe on more than 1 variable...

```
data2_sorted<- data2[order(data2$year, data2$len),]
data2_sorted
```

Now you should be able to get R to do most anything you want it to with data!

SPECIAL AND USEFUL MATRIX FUNCTIONS

Matrices are the backbone of most statistical analyses and are an essential tool in the quantitative ecologists toolbox. Here we cover a few neat tricks with matrices that may come in handy later.

```
### First create a 4 row by 5 column matrix
MTX <- matrix(c(1,1,1,1,1,2,2,2,2,2,3,3,3,3,3,4,4,4,4,4), ncol
= 5, byrow = T)

# Print it out
MTX

## Transpose the matrix
# notice rows are columns and columns are rows
t(MTX)

# create a scalar
A <- 0.5

# multiply the matrix by a scalar
MTX*A

# multiply matrix by another scalar
MTX*10

## create a vector
V = c(10,1,0.1,0.01)

## multiply the matrix by a vector
## WARNING THIS IS NOT MATRIX MULTIPLICATION
MTX*V
```

```
# To illustrate matrix multiplication we first create an
identity
# matrix. This is a matrix with 1's along a diagonal from the
top
# left to bottom right

IDENT = matrix(c(1,0,0,0,1,0,0,0,1), ncol= 3)
IDENT

# Identity matrices are super useful in quantitative
applications so
# you've probably used them and didn't know, Ok, next we
create
# a vector that has the number of elements equal to the number
of
# columns in the matrix

V.new = c(1,2,3)

# matrix multiplication is specified using %*%
# so we have

IDENT %*% V.new

## it should have returned a column vector 1,2,3
# what happens if we just use the regular multiplication
operator, *

IDENT * V.new
```

This is because matrix multiplication works this way

$$\begin{bmatrix} 2 & 5 \\ 3 & 6 \\ 4 & 7 \end{bmatrix} \times \begin{bmatrix} 10 \\ 15 \end{bmatrix}$$

2*10 + 5*15 = 95

3*10 + 6*15 = 120

4*10 + 7*15 = 145

results in the vector:

$$\begin{bmatrix} 95 \\ 120 \\ 145 \end{bmatrix}$$

```
## check with a little R code
c = matrix(c(2,3,4,5,6,7), ncol = 2)
z = c(10,15)
c %*% z
```

whereas regular multiplication with matrices in R works this way

$$\begin{bmatrix} 2 * 10 & 5 * 15 \\ 3 * 15 & 6 * 10 \\ 4 * 10 & 7 * 15 \end{bmatrix} = \begin{bmatrix} 20 & 75 \\ 45 & 60 \\ 40 & 105 \end{bmatrix}$$

```
## check with a little R code
c * z
```

Why should we care? Well, matrices and matrix multiplication is one of the essential tools for the quantitative ecologists- in particular we use them to model populations with Leslie / Lefkovitch matrices Consider a simple population with age/ stage classes:

```
# per capita fecundity juvenile
Fj = 1.1
# per capita fecundity adult
Fa = 2.1

#survival age 0
S0 = 0.25
#survival juvenile
Sj = 0.35
#survival adult
Sa = 0.75
```

We create the population transition matrix

```
trans.mtrx = matrix(c(0,Fj,Fa,S0,0,0,0,Sj,Sa), ncol = 3, byrow
= T)
#print it out
trans.mtrx
```

We create the population vector

```
#Number of animals in each age class
N0 = 100
Nj = 50
Na = 200
```

```
Nt = c(N0,Nj,Na)

## whats the population estimate for next year?
trans.mtrx %*% Nt
```

R has several built-in functions for manipulating and evaluating matrices for population models we can perform an eigen analysis using eigen() function the first element in the list created by the function is the eigenvalue also known as lambda, the population growth rate lambda:

```
#eigen analysis of population transition matrix
eigen(trans.mtrx)$values[1]

## stable age distribution
eigen(trans.mtrx)$vectors[,1]/sum(eigen(trans.mtrx)$vectors[,1
])
```

One more built-in function to show you grabs the diagonal elements of a matrix diag, it could come in handy later, and using the variance covariance matrix from a linear model

```
# we'll use the identity matrix
IDENT
# grab the diagonal elements
diag(IDENT)
```

WORKING WITH DATES, FOR-LOOPS AND IF-ELSE

Working with dates

Dates are the last type of object that we will be using in this course. We save this one for last because they can be a real pain to work with, well… at least for one of us. Before we begin, download the file **dates.csv** to your working directory. It is a comma separated text file. Open it in your favorite text file reader (e.g., notepad, excel, word) and examine. Create a data frame “dater” by reading dates.csv into R.

```
## don’t forget to set YOUR working directory
setwd("C:/Users/Jims XXXXX ")

## read example data file into a data frame
dater<-read.csv("dates.csv")

## let’s see what is in the data frame
head(dater)
```

Just like other objects, we can query the objects (think: columns) in the data frame using the class function. For example,

```
## what are the type of objects in the data frame
class(dater$date.m.d.y)
```

Hmmm… let’s think a bit about this. You’ve use the “is.” function to find out if something was numeric, is.numeric, and character, is.character. We also learned that we can coerce objects using the “as.” function. What can we do to change date.m.d.y into a date? If you guessed as.Date you get a gold star. The as.Date function coerces character and factor objects into dates based on their syntax. Let’s looks at the syntax of date.m.d.y.

```
#print out columns
dater$date.m.d.y
```

You should have noticed that the values were arranged in *month*, *day*, *year* order and that the values were delimited using a slash “/”. You need to specify the correct order and the delimiter in the as.Date function using the “format = “ option. The function uses specific codes to refer to the format of the day, month, and year shown in the table below

Code	Value
%d	Day of the month (decimal number)
%m	Month (decimal number)
%b	Month (abbreviated)
%B	Month (full name)
%y	Year (2 digit)
%Y	Year (4 digit)

We see from the table that %d represents day, %m is the decimal number for month, and %Y is used for year in the 4 digit format, e.g., 2013. OK now we're ready to specify the format of the date in date.m.d.y. We have month in decimal format, day, and year in 4 digit format, so format="%m/%d/%Y". Now we have:

```
#create Date1 using values in dater$date.m.d.y
Date1 <- as.Date(dater$date.m.d.y, format="%m/%d/%Y")
# print it out
Date1

# first coerce dater$date.m.d.y to become character
# then coerce to become date
Date1 <- as.Date(as.character(dater$date.m.d.y),
format="%m/%d/%Y")

#check on the type of object we created
```

```
class(Date1)
```

Ok, what if my data are in day, month, year format separated by a forward slash “/”. It just so happened that the data in `dater$date.d.m.y` are in that format. We have day, month in a decimal format, and year in a 4 digit format all separated with a forward slash To coerce into a date we should:

```
# first check the class of the object (this is good practice)
class(dater$date.d.m.y)

# create Date2 by coercing dater$date.d.m.y notice the format
is day
# month year
Date2 <- as.Date(dater$date.d.m.y, format="%d/%m/%Y")
#print it out
Date2
# check the class again for grins
class(Date2)
```

Try coercing the remaining objects in dater.

```
##First let’s try dater$date.y.m.d print it out and examine
dater$date.y.m.d

## looks like year (4 digit) month, day try coercing using the
code below
as.Date(dater$date.y.m.d, format="%Y/%m/%d")
```

What happened when you submitted the above code? You should have obtained a bunch of NA and some nonsense dates. This is because we mis-specified the delimiter. We used a forward slash instead of a dash or minus sign “=”. This type of output, wrong dates and NAs are usually a good sign that you mis-specified the date format. Let’s do it correctly this time using a dash for the delimiter.

```
# create Date3 by coercing dater$date.y.m.d notice the use of
dashes
Date3 <- as.Date(dater$date.y.m.d, format="%Y-%m-%d")

### print out next object. Notice the values are delimited
using a period
dater$date.d.m.y.2

# create Date4 by coercing dater$date.y.m.d notice the use of
periods
Date4 <- as.Date(dater$date.d.m.y.2, format="%d.%m.%Y")

### print out next object dater$date.m.d.y.2.
dater$date.d.m.y.2
```

Notice the values in date.m.d.y.2 are delimited using dashes also that the format for month is now an abbreviation. In the table above, the abbreviated month format is %b, so we change the format in the `as.Date` function like:

```
# create Date5 by coercing dater$date.m.d.y.2 notice the use
%b and dashes
Date5 <- as.Date(dater$date.m.d.y.2, format="%d-%b-%Y")
```

Once dates are in the proper format, you can perform a variety of R functions, such as summarizing data by date, calculating the mean, median, minimum and maximum dates (more next) or simple functions like addition or subtraction. For example,

```
# difference in days between Date1 and Date2
Date1 - Date2
```

Another useful function is `difftime`, which calculated the difference between 2 dates. Let’s see what this function can do using the help function.

```
help(difftime)
```

Here we see that we can specify the difference units in terms of days, weeks, hours, and seconds. For example, we can calculate the difference between Date2 and Date1 in terms of weeks.

```
difftime(Date1,Date2, units = 'weeks')
```

Hmmm... you’re wondering. Why can’t I simply divide the difference between Date2 and Date1 and divide by 7? Go ahead, try it and find out what happens (don't be sceered): e.g.,

```
 (Date1 - Date2)/7
```

Now, why would I want to calculate intervals between dates? Well maybe you are going to fit a hazards model or maybe you need to calculate a common time interval for an open population capture-recapture model. There are several reasons to do this in ecological studies, so it’s important to know that this is possible.

Quite often, we want to extract a part of a date, such as a year, or a month. We can do this using the format function as shown below. Be aware, however that the objects produced by format are characters, so they need to be coerced if you want them in numeric format, e.g.,

```
# create year by stripping the year value in Date1 and make it
numeric
year <-as.numeric(format(Date1, format = "%Y"))

# create month by stripping the month value in Date1 and make
it numeric
month <-as.numeric(format(Date1, format = "%m"))

## create month.char by stripping the month value in Date1 and
output
## in abbreviated character format
month.char <- format(Date1, format = "%b")

# create day by stripping the day value in Date1 and make it
numeric
day <-as.numeric(format(Date1, format = "%d"))
```

We can also create on object with day of year (1-365) a.k.a. *Julian date* using `strftime` and specifying `format = “%j”`. As above, this object must be coerced to make it numeric:

```
## create Julian date from Date1 and coerce into numeric
Julian<-as.numeric(strftime(Date1, format = "%j"))
```

The strftime function has several other uses, such as extracting hours n=minutes and seconds from objects with date time values. To find out more, we use the help function.

Ok so we can coerce a date from a character or factor format and we can extract parts of dates. *What about creating a date from objects containing day, year, and month?* Fortunately, this is easy using the ISOdate function. For example, we can create a date object using the year, month, and day objects.

```
# create object my.date using year, month, and day created
above
my.date<-ISOdate(year,month,day)
#print it out
my.date
```

The order of the year, month, and day inside the matters in the `ISOdate` function, so you should always check the function format and options using help. You should have noticed that `ISOdate` creates an object that contains times too. This is useful if you have date and time data, but if you do not it can be annoying. We can reformat my.date using `strptime`. The format should match the year month day format of my.date:

```
## just extract the year  month and day from my.date
strptime(my.date, "%Y-%m-%d")
```

There are several other neat tricks that we learned earlier working with characters and numbers that will also work with dates. For example, we can create a sequence of dates that begin with June 1, 2000 to August 8, 2000, for a specified interval using the seq function:

```
#output dates every 2 weeks
seq(as.Date('2000-6-1'),to=as.Date('2000-8-1'),by='2 weeks')

#output dates each day
seq(as.Date('2000-6-1'),to=as.Date('2000-8-1'),by='1 days')
```

Here we have only scratched the surface of what can be done with dates in R. This should be enough for most applications. If not, you will need to look into using one of several available packages, such as the **date** and **lubridate** packages.

Comparison (logical) operators

When manipulating data or simulating ecological processes, we often need to use a comparison operator to compare two or more values and performing an operation if values meet certain specifications. We do this using an “if” statement. The first step is to determine what sort of comparison we want to make. Let’s say I want to know if the value of an object, abundance, is greater than, say 0.

```
# create abundance and assign value of 10
abundance = 10

# determine if abundance is greater than zero
abundance > 0
```

If the condition is met the comparison operator will return TRUE. Next we want to create a new variable, present and assign it a 1 if abundance is greater than 0 using an if statement

```
# make comparison
if(abundance > 0) present = 1
## print out
present
```

What do you think would have happened if abundance was equal to zero? Go ahead and try it out, but first we need to remove the present object using….. come think back….. the remove or rm function

```
#remove present
rm(present)

#set abundance to zero
abundance = 0

# make comparison
if(abundance > 0) present = 1
## print out
present
```

What happened? Well if you did it right you should have gotten the following error message:

```
Error: object 'present' not found
```

This is because the condition in the if statement was not met (= FALSE) so the action that came after the if statement was not executed (it wasn't run). We need a way to modify the if statement to include what else to do if the comparison is FALSE. What else to do, what ELSE to do? How about use an else statement? Let's try:

```
# make comparison
if(abundance > 0) present = 1 else present = 0
## print out
present
```

The above is the basic format for the if-else statement. However, it does not cover the instances where you want to do multiple things if the condition is, or is not met. To do that, we need to use curly brackets “{ }” to delineate the actions we want to accomplish. For example, let’s say that we want to create another variable occupied = 1 if abundance is greater than 0 and occupied = 0 if abundance less than or equal to zero.

```
# make comparison
if(abundance > 0){ present = 1
                   occupied = 'yes'
} else{ present = 0
      occupied = 'no'
      }
## combine and print out
c(present, occupied)
```

Change abundance equal to 10 and re-run the above code. Notice that the program executes everything inside the brackets depending if the comparison is TRUE or FALSE. What if we had 2 comparisons to make. For example, let’s say we wanted to create a variable season based on the months of the year. First create a date object `new.date` as 5/13/2015:

```
new.date = as.Date("5/13/2011", format = "%m/%d/%Y")
```

Now let’s assign season = spring if month is May to June. So think about it.... May is the 5th month and June is the 6th month, so if month is between (not including) 4 and 7 the day in new.date should be in the spring. We can use the "&" operator to select the instance where month is greater than April (month 4) and less than July (month 7):

```
### make comparison and assign season
if(as.numeric(format(new.date, format = "%m"))> 4 &
as.numeric(format(new.date, format = "%m")) < 7) { season =
'spring'
## if its not spring it is another season
} else{season = 'other'}

#print it out
season
```

Now sets assign a new date to new.date as Feb 11 and assign season = winter if the month is between November and April.

```
new.date = as.Date("2/11/2013", format = "%m/%d/%Y")

# just modify the above code
if(as.numeric(format(new.date, format = "%m"))< 4 &
as.numeric(format(new.date, format = "%m")) > 11) { season =
'winter'
} else{season = 'other'}

#print it out
season
```

Did that work? Hopefully you said NO! That's because we used the & and asked if the month was less than 4 (April) AND greater than 11 (November). What we really want to ask is if the moth is was less than 4 (April) OR greater than eleven. The OR is symbolized by the pipe "|" so we have:

```
if(as.numeric(format(new.date, format = "%m")) < 4 |
as.numeric(format(new.date, format = "%m")) > 11) { season =
'winter'
## assign other to non-winter months
} else{season = 'other'}

#print out
season
```

There are several operators for making comparisons in R and these are shown the table below

Operator	Description
<	**less than**
<=	**less than or equal**
>	**greater than**
>=	**greater than or equal to**
==	**exactly equal**
!=	**Not equal to**
x|y	**x OR Y**
x & y	**x AND Y**

Now let's try some comparisons using the Date1 vector. First, let's see which dates in Date1 fall in the second half of the month, that is, the days greater than or equal to 15.

```
# identify days greater than equal to 15 (TRUE)
as.numeric(format(Date1, format = "%d")) >= 15
```

Now let's create an object month.part that is assigned a value of 2 if the day is in the second half of the month and otherwise equals 1 using the if else function.

```
# identify days greater than equal to 15 (TRUE)
if(as.numeric(format(Date1, format = "%d")) >= 15) month.part
= 2 else month.part = 1
```

What happened after you submitted the statements? You should have gotten the following error:

```
Warning message:
In if (as.numeric(format(Date1, format = "%d")) >= 15) :
  the condition has length > 1 and only the first element will
be used
```

This is because we tried to use a scalar function if else of a vector. In other words, i**f-else functions** can only make comparisons on one value at a time. For example, recall that we can use the bracket notation [] to refer to specific elements in a vector. For example, the first date in Date1:

```
Date1[1]
```

We could use that notation to make the comparison one at a time. For example

```
# identify if first day greater than equal to 15 (TRUE)
if(as.numeric(format(Date1[1], format = "%d")) >= 15)
month.part = 2 else month.part = 1
```

We could do that as many times as there are elements in Date1 one at a time. This would be a real pain for large datasets and would make for very large programs. This also is a repetitive task. As you may have guessed, there is a function that we can use to perform repetitive tasks, **for loops**.

For loops

The basic syntax of a for loop in R is

```
for(i in min:max) {
                   do something max minus min times
                  }
```

The for loop is delineated by brackets and anything inside of the brackets it executed the number of times specified by the min and max. For example, lets assign a value of 10 to an object Y and add 2 to it for 10 iterations

```
Y = 10
for(i in 1:10) {
               # add 2 to Y
               Y = Y + 2
               # print it out?
                Y
               }

#print it out last value
Y
```

Notice that the for loop does not print out each of the 10 steps to the console. To do that, we need to add a print statement like below.

```
Y = 10
for(i in 1:10) {
                # add 2 to Y
                Y = Y + 2
                # print it out!!
                print(Y)
                }
```

Now we able to see the changes in Y through each step. What about saving the values of Y in an object. That should be easy. We simply declare a vector variable Z and add values of Y to Z at each step. For example,

```
Z=c()
Y = 10
for(i in 1:10) {
               # add 2 to Y
                Y = Y + 2
                # save values of Y each step
                Z = c(Z,Y)
               }
# print it out
Z
```

For loops really come in handy when we do ecological simulation at the end of the workshop. For example, let's take an initial population size (N) of 100 and model populations for 10 years assuming a population growth rate λ = 1.05

```
#set initial population size
N = 100
# create a place to put the simulated time series data
# star with year = 0 and initial population size N
time.series=c(0,N)
## growth rate
lambda = 1.05
# Yearly time step loop for 10 years
for(year in 1:10) {
                  # grow the population
                  N = N*lambda
                  # save values
                  time.series = rbind(time.series,c(year,N))
                  }
# print it out
time.series
```

Notice that year increases from 1 to 10 and that year is an R object known as the **for loop index**. We can actually use the **for loop index** in calculations or other ways an scalar value is used. For example, we can use the index to refer to a specific column or row in a matrix or vector. Let's say that we wanted to compare the population size from one year to the next in our time.series. This would entail comparing values in one row with values in an adjacent row. The script below does just that. It compares values in row i with values in i-1 and assigns a value to trend. Run the script.

```
## create plane to hold trend assessment
trend= c()
for(i in 2:11){
            ## population size is in column 2
            ## compare pop size to previous pop size and
create trend object
             if(time.series[i,2] > time.series[i-1,2])
trend[i] = "increasing" else trend[i] = "decreasing"
}
# print it out
trend
```

Getting back to our motivation for modeling repetitive tasks, we originally wanted to create an object month.part that is assigned a value of 2 if the day in Date1 is in the second half of the month and otherwise equals 1 using the if else function. To do, this we could use a for loop like we did with the time.series data and use the **for loop index** to refer to the individual values in Date1:

```
#how many elements in Date1
length(Date1)

# create the object month.part
month.part = c()

for(i in 1:length(Date1)){
  # identify if first day greater than equal to 15 (TRUE)
  if(as.numeric(format(Date1[i], format = "%d")) >= 15)
month.part[i] = 2 else month.part[i] = 1
  }

# print it out
month.part
```

Well that worked! It was a bit clunky but demonstrates one of the many uses of for loops. We will be using for loops for the remainder of the course so you will get plenty of opportunities to use them and learn more tricks-o-the trade. But, here’s one quick one before we leave this subject today. We mentioned that for loops execute until the index reaches the maximum value indicated in the parentheses. There are times that we want to escape from a for loop before it finishes. For example, let’s say were simulating a population and it goes extinct, N = 0. We don’t want to continue simulating because nothing is going to happen, so we need to jump out of the for loop. Here’s one way to do that using break.

```
## initial population size
N = 10
## create place for population time series
popn = c(0,N)
## population growth rate
lambda = 0.85
# for loop with index year
for(year in 1:100){
                  N = N*lambda
                  # save population data
                  popn = rbind(popn,c(year,N))
                  # if population size is less than 1 break
out of loop
                  if (N < 1) break
                  }
# print it out
popn
```

Vector operators

The preceding use of a for loop to make comparisons with vector data was a bit clunky and with large datasets it would be slooooow. A more efficient was to make comparisons is by using a vector operator. Here the function is ifelse The syntax for ifelse is:

```
ifelse(comparison, value if true, value if false)
```

Going back to our previous example, we have:

```
ifelse(as.numeric(format(Date1, format = "%d")) >= 15, 2,1)
```

That is much more efficient that creating for loop to make comparisons. R has several matrix and vector operators that are faster and will make your programs shorter and easier to code and run. We will learn more about some of these later.

COUNTS, MEANS, SUMS, RANKS, AND RELATED INFORMATION

Summarizing data

Be sure to have the following 3 comma separated text files:

Fish.catch.csv: contains fish collection data from 8 streams as part of the Coweeta LTER

stream.habitat.csv: habitat measurements made concurrently with fish collections

Water.measures.csv: water quality measures made concurrently with fish sampling

Read the files into R and examine. Be sure to set your working directory to the proper location.

```
## set working directory
setwd("C:/Users/Jim/Desktop")

# read in commas separated files
catch<-read.csv("Fish.catch.csv")
habitat<-read.csv("stream.habitat.csv")
water<-read.csv("Water.measures.csv")

#whats in these data frames
head(catch)
head(habitat)
head(water)
```

Data summaries are always a good way to check your data prior to any analysis. R has several built-in functions for performing data summaries. Below we have several functions that you can use to summarize values in numeric vectors. Note that the result of all of these functions can be output to R objects. Before we try a few, let's find out what objects are in the water data frame using the names function.

```
## what are the variables in the water data frame
names(water)

mean(water$Water_Temp)
median(water$Water_Temp)
sd(water$Water_Temp)
var(water$Water_Temp)
min(water$Water_Temp)
max(water$Water_Temp)
sum(water$Water_Temp)

## how many elements are in this object?
length(water$Water_Temp)

## complete summary of data in water
summary(water)

## what are median 50% and the lower and upper 95%
# quantiles of water temperatures
quantile(water$Water_Temp,probs=c(0.025,0.5,0.975))
```

```
# note that you can specify other quantiles by putting
different
# he we specify the 20, 40, 60, and 80th percentiles
quantile(water$Water_Temp,probs=c(0.2,0.4,0.6,0.8))
```

We could combine each one of the objects into a single vector. For example:

```
## combine mean, standard deviation(sd), min and max of water
temperature
water.temp<-
c(mean(water$Water_Temp),sd(water$Water_Temp),min(water$Water_
Temp),max(water$Water_Temp))

## combine mean, sd, min and max of conductivity
conduct <-
c(mean(water$Conductivity),sd(water$Conductivity),min(water$Co
nductivity),max(water$Conductivity))

## combine mean, sd, min and max of dissolved oxygen
DO<-c(mean(water$DO),sd(water$DO),min(water$DO),max(water$DO))

## Create a table with the combined summary data using rbind
function
sum.data <- rbind(water.temp,rbind(conduct,DO))

#print it out
sum.data

# what type of object is it?
class(sum.data)
```

We also could combine the sum.data with a vector containing the names of the variables. Give it a try on your own. Don't forget that a matrix cannot contain character and numeric values only a data frame can.

Now that (above) was a bit wasteful. We had to perform the same operation 3 times, once for each object (column) in the data frame. We could be more efficient if we create a for loop to perform the operations. Remember that we can subset data in a data frame using numbers for the rows and columns. In the water data frame, data in columns 3 to 6 contain the measurements that we are interested in summarizing (Water_Temp, Conductivity, DO, Turbidity). If we wanted to calculate the mean of Turbidity, it is in column 6 so:

```
## calculate mean Turbidity in column 6
mean(water[,6])
```

If you recall, we can use the **for loop** index (the number) to select rows or columns using their numbers. Below is a for loop for creating a matrix containing the mean, standard deviation, minimum, and maximum for Water_Temp, Conductivity, DO, Turbidity (columns 3 to 6). Examine it carefully, copy it to your R script and run.

```
## save the names of the columns in a file, water.name
water.name = names(water)

## create place to put the summary data
sum.table = c()
```

```
## for loop for calculating and combining summary data by
column
for(i in 3:6){
  ## calculate summary statistics for each column from 3 to 6
  ## use rbind to stack the values into a single matrix
  sum.table<-
rbind(sum.table,c(mean(water[,i]),sd(water[,i]),min(water[,i])
,max(water[,i])))
}
```

To create a nice table, we combine summary data matrix sum.table with the names of the water columns we used. These names are elements 3 to 6 in water.names. Note that we are combining characters and numbers so we must convert the names to a data frame before the cbind. This is why as.data.frame(water.name[3:6]) is inside the cbind function.

```
## combine variable names with corresponding summary
statistics
sum.table <-cbind(as.data.frame(water.name[3:6]),sum.table)

## the columns need names, so let's do it
colnames(sum.table) =
c("Characteristic","Mean","SD","Min","Max")
```

Creating a data summary table like we did above is probably not a onetime thing that you will do. In fact, we suspect that most of you will be creating data summaries throughout your careers. Wouldn't it be wonderful if you didn't have to go through all of those steps each time you wanted to create a similar table? You could if you created a function. So far you have been using R built in functions, but you don't have to. You could create your own. It's actually very easy in R. For grins, let's create our own function for calculating a mean and call it my.mean. Remember that a mean is the sum of the elements in a data divided by the number of elements in a data. Looking at the functions listed above, we use `sum()` to add up the elements and length() to could the number of elements with the mean equal to `sum()/length()`.

Creating functions

To create a function, we declare it using `function` and assign the function a name. Similar to **for loops** and **if else** comparisons, function actions are delimited by curly brackets "`{}`". The function can take one or more *arguments* (these are the inputs to the function). For instance, `my.mean` (below) takes one argument: variable. If there are more than one input to a function, ***THE ORDER THEY ARE LISTED MATTERS*** (mas later).

```
## create a function for calculating the mean
## first name it and identify arguments
my.mean <- function(variable){
     # users of the function provide “variable” and this is
what is done with it
     sum(variable)/length(variable)
     }

## use the new function
my.mean(water$Water_Temp)

## compare to built-in R mean function
mean(water$Water_Temp)
```

Our function for calculating the mean is spot on. Now, let’s create something that requires more than 1 input, remember ***ORDER MATTERS***. Examine the code below and interpret.

```
## create a function for dividing 2 numbers and squaring the
result
my.add.square <- function(a,b){
             (a/b)^2
             }
# use the function
my.add.square(5,2)

# use the function but reverse the numbers
my.add.square(2,5)
```

For grins, look at the contents of your working directory. Can you find “a” and “b” objects? The answer should be ***no***, because R functions create and use ***local variables***. This means that *anything created in a function is not saved to your working directory*.

The above demonstrates the usefulness of functions and the idea that the order of the inputs matters. Now let’s try to create a function that creates a summary table similar to the one we created above and call it mk.sum.table. Remember, this a learning experience so we may not get it right the first time. In fact, we usually create the steps used in a function first (i.e., before enclosing them in function brackets) because it is easier to diagnose potential problems with code. We’ve already done that above, so here we simply clean things up a bit and put them inside of a function. Here we go:

```
## Lets create a function that creates the summary table so we
can use it later
## inputs are name of data frame (dat.frm), and column number
of first variable to
## summarize note we will assume that we will summarize all
variables from that
```

```
## column number to the end of the data frame.
mk.sum.table <- function(dat.frm,first){
  # save names in columns to local file
  namz = names(dat.frm)
  #create place to put summary data
  tablez = c()
  ## calculate summary statistics for each column from first
to last
  ## last is calculated using length() function applied to
column names
  for(i in first:length(namz)){
    ## use rbind to stack the values into a single matrix
    tablez<-
rbind(tablez,c(mean(dat.frm[,i]),sd(dat.frm[,i]),min(dat.frm[,
i]),max(dat.frm[,i])))
  }

  ## combine summary data with column names in namz
  ## note that your combining characters and numbers so we
  # must convert the names to a data frame before the cbind
  tablez <-
cbind(as.data.frame(namz[first:length(namz)]),tablez)

  ## the columns need names
  colnames(tablez) =
c("Characteristic","Mean","SD","Min","Max")

 ## this returns the data frame that was created otherwise
nothing is output
  return(tablez)
}
```

```
## now let’s invoke the function
mk.sum.table(water,3)

## let’s try to summarize the habitat file
## first, see what in it
head(habitat)

## try the new function with habitat
## first column to include is 4- Length
mk.sum.table(habitat,4)
```

Something went wrong above. What was it? Before freaking out... (*learning a programming language is as much about failure as about success*), let’s see if we can calculate a simple mean on the 4th column:

```
mean(habitat[,4])
```

The function should return an NA. This is because the habitat data contain missing values! That was the problem. While we’re here…

Commonly Encountered Problems (CEP) with built in functions: The default settings for many built-in R functions cannot deal with missing values and will return errors or NA, NaN, or the dreaded Inf. Users must specify how to handle missing data. For example, most summary functions, such as mean and sd. Require that you specify that missing values be removed using “na.rm = TRUE”, e.g.,

```
## here we can tell the function to remove the missing data
mean(habitat[,4], na.rm = TRUE)
```

To get the mk.sum.table function to work, we simply add “na.rm = T” (short hand for “na.rm = TRUE”) to each of the summary functions. e.g.,

```
## Modify the above function by adding na.rm to each function
mk.sum.table <- function(dat.frm,first){
  namz = names(dat.frm)
  tablez = c()
  for(i in first:length(namz)){
    ## calculate sumary statistics for each column
    ## use rbind to stack the values into a single matrix
    tablez<- rbind(tablez,c(mean(dat.frm[,i],na.rm =
T),sd(dat.frm[,i],na.rm = T),min(dat.frm[,i],na.rm =
T),max(dat.frm[,i],na.rm = T)))
  }

  ## combine summary data with column names in namz
  ## note that your combining characters and numbers so we
  # must convert the names to a data frame before the cbind
  tablez <-
cbind(as.data.frame(namz[first:length(namz)]),tablez)

  ## the columns need names
  colnames(tablez) =
c("Characteristic","Mean","SD","Min","Max")

 ## this returns the data frame that was created otherwise
nothing is output
  return(tablez)
}

## try the new function with habitat
```

```
# first column to include is #4 Length
mk.sum.table(habitat,4)
```

The function works! Now we can summarize all kinds of data.

We could have avoided the problems with missing data above if we had examined the data first. For example we could have used the summary function to see if the data contained NA.

```
summary(habitat)
```

There are several ways to handle missing data when conducting statistical analyses or fitting models. One way is to eliminate missing data and as we learned earlier we can use na.omit to select a data frame with no missing data. Another method is to replace NA with some default value, usually the mean value. Thinking back we can use the ifelse function to make a comparison in a vector and replace values if the comparison is true or false.

Remember that we can use the “is.na” function to identify where data were missing

```
#identify the missing length data
is.na(habitat$Length)

#calculate the mean length for use below
mean(habitat$Length,na.rm = T)

Let’s put these things together and replace missing data,
is.na = TRUE, with the mean calculated above, otherwise (else)
if not missing keep the non-missing value
```

```
   #replace the missing data with the mean length
   ifelse(is.na(habitat$Length),11.479,habitat$Length)
```

We could get really fancy and put the mean function inside of the ifelse function.

```
## replace missing length data but put function inside ifelse
function
ifelse(is.na(habitat$Length),mean(habitat$Length,na.rm =
T),habitat$Length)
```

Performing summaries by groups

Awesome! Thus far we learned how to summarize data in vectors and elements on a data frame. More often than not, we wish to summarize data by some groups, say by year or study site or geographic region. To do this, we need to learn how to use array (matrix, list) operators. That is, operators that use data in more than one column. One useful function is tapply. Use the help() function to examine the tapply syntax. Below we use tapply to calculate mean sample unit length (Length) by habitat type (Habitat) in habitat data frame.

```
   ## first examine contents of habitat
   head(habitat)
```

Now we use tapply listing (*in order*): the variable to summarize (or analyze) , the grouping variable, the function to use (hear the mean) and function options (here remove NA):

```
## use tapply to summarize by groups
by.hab.mean <- tapply(habitat$Length,habitat$Habitat, mean,
na.rm = T)
#print it out
by.hab.mean
```

If we want to use the object created by tapply, it's important to know what type of object was created.

```
  # what kind of object is it
  class(by.hab.mean)

  str(by.hab.mean)

  by.hab.mean[1]
```

Here, we see that by.hab.mean is an array that consists of 3 numeric variables and 3 headings (character variables). To put this in a more manageable and familiar form, we can coerce the list into a data frame.

```
  ## coerce into a data frame
  as.data.frame(by.hab.mean)
```

One thing about this coercing an array is that the headings were turned into ***rownames***. As far as we're concerned, *rownames* are useless because you can't use or manipulate them like the elements of a data frame, so let's extract the rownames and put them in a column in the data frame:

```
## coerce into a data frame with row names now in a column
data.frame(rownames(by.hab.mean),by.hab.mean)
```

We end our discussion of tapply here because there is a better and more useful function for summarizing data: aggregate, which requires users to specify the columns to summarize and the groups using a list if column numbers inside of brackets. For example, columns 4 and 5 in the habitat data frame contain Length and Width measurements of sample units and the 3rd column of habitat contains habitat type so to calculate summary statistics for Length and Width “habitat[c(4,5)]” by habitat type “by = habitat[c(3)]”, the syntax of aggregate is:

```
### a new function for summarizing data by groups
## values inside [c()] correspond to column numbers
aggregate(habitat[c(4,5)], by = habitat[c(3)],mean,na.rm = T)
```

Copy the code and paste in R script, run and examine the output. We can aggregate by more than 1 group using the aggregate function. For example, to get means by stream and habitat type, "by = habitat[c(1,3)]”. Copy the code below to your R script and examine. Interpret the code, run and examine the output.

```
## we can do by more than 1 group for example stream and
habitat
aggregate(habitat[c(4,5)], by = habitat[c(1,3)],mean,na.rm =
T)

## lets create an object with all of the means
hab.means <-aggregate(habitat[c(4:9)], by =
habitat[c(1,3)],mean,na.rm = T)
```

```
## what kind of object did we create
class(hab.means)
```

As you can see, aggregate is very useful and produces an object (a data frame) that is easy to work with.

Summaries with categorical data

The previous sections dealt with summarizing numeric data. Here, we cover how to summarize categorical data (characters, strings, and factors). For example, you may want to count up the numbers of times that a species was caught during your research. To demonstrate, we will use the catch data read in earlier. First, examine the (partial) contents of the data frame.

```
head(catch)
```

You should have noticed that the data consisted of electrofishing catch data from stream habitats. Each fish that was collected was entered in the database along with the fish's total length (TL.mm) and the place date and habitat type where it was captured. The built-in function table is useful for summarizing categorical data and will count up the number of instances that a category occurs in the data frame or vector. For example, the code below counts up the number of times each species (catch$Species) occurs in the data frame. Examine the code, run it, and examine the output.

```
## new function for summarizing data
spc.collect <-table(catch$Species)
spc.collect
```

The numbers below each species name is the number of times the species is listed in catch$Species. Because each fish that was captured is listed by species, this number corresponds to the total number of individual fish of that species that were captured. This number could be used in an analysis so let’s see what kind of object is spc.collect.

```
## that type of object is this
class(spc.collect)
str(spc.collect)
```

Hmmm, it’s a table…another type of R object (similar to an array) that has limited usefulness for further analyses. Below, we will coerce the table into a data frame and rename the columns. Examine the code and interpret each step, then run the code and examine the output.

```
spc.collect<- as.data.frame(spc.collect)
spc.collect
colnames(spc.collect) = c("Species","Total.catch")
spc.collect
```

The spc.collect data frame we created could be written to an external file and included in a thesis or report or could be used for further analysis in R.

As you may have guessed, we can include more than one column (or object) in a table function. For example, the below code counts up the number of instances of each combination of species (catch$Species), stream (catch$Stream), and habitat type (catch$Habitat). Examine the syntax. Notice that table function is nested within the as.data.frame function. This means that table will create an object that will be coerced into a data frame and then written to the data frame tot.catch. Run the code and examine the output.

```
## create an object that
tot.catch<-
as.data.frame(table(catch$Stream,catch$Habitat,catch$Species))
## print it out
tot.catch
```

You should have noticed that table includes values (zeros) where there were no instances of various combinations of the species, stream and habitat type. For example:

```
1   Howards Branch   Pool          Blacknose Dace    0
2         L Ball Crk   Pool          Blacknose Dace    0
3   L Drymans Fork   Pool          Blacknose Dace    0
4         M Ball Crk   Pool          Blacknose Dace    0
5       Sheppard Crk   Pool          Blacknose Dace    0
```

These are instances where the data contains no observations of Pool habitat type and Blacknose Dace at these 5 streams. Be absolutely sure that you understand that table will produce summaries that include combinations of the variables that were never observed in the data frame. This may be desirable, for instance-- when you want to know samples or locations where a species was not collected for occupancy modeling, or may not be desirable. Just be sure that you know what you want.

Can you think of a way we could eliminate the zero observations?

Ok, lets create a similar table (as above) using the aggregate function. Here we are counting up the number of elements of TL.mm (the number of fish) by stream, habitat, and species: Examine the code and try to interpret. Run the code and examine the output.

```
## lets create a similar object using aggregate and counting
the number of
## elements using length
tot.catch.nozero<- aggregate(catch[c(5)], by = catch
[c(1,3,4)],length)
#print it out
tot.catch.nozero
```

You should have noticed that the code produced a data frame with the total number of fish collected at each site and habitat type, by species but with no zeros. (Unfortunately, the heading to the column is TL.mm – this would have to be changed to Total.collected or similar to avoid confusion).

Put your thinking caps on. Let’s say that we needed to calculate the total number of species captured in a sample. *How could we do this?*

There are probably a large number of ways to do this. What we need is a data frame with the species that were collected at each site, date, habitat type. The catch data frame contains multiple observations corresponding to individual fish. We need to reduce the data frame down to just the species that were collected. We can do this using the unique() function. Here we want the unique combinations of stream, date, habitat type, and species, columns 1 to 4 in catch. Examine the code below and submit to R.

```
# create species using unique combinations of stream, date,
habitat type, and species
species<-unique(catch[,1:4])
```

```
# print it out
species

# create a new object (column in species) and assign a 1 to
species that were
# captured aka detected
species$detect = 1

## calculate richness but summing the total number of species
detected in a sample
richness<- aggregate(species[c(5)], by = species[c(1:3)],sum)
```

We now have the total number of species detected (aka species richness) in the detect column. Let’s see what habitat variables were correlated with species richness. We first need to combine the richness data with the habitat data using the merge() function. Note that fish may not have been collected (catch = zero fish) in all samples, so we use the "all = TRUE" option to include all stream, dates, and habitats. **Reminder**: the all option in merge() indicates that all of the observations in both data frames should be included in the new data frame.

```
# create new combined data frame
for.correl <- merge(richness,habitat, all = TRUE)
# print it out
for.correl
```

After examining the contents of for.correl, you should see that richness estimates (the detect column) are NA for several samples. There are the samples where no fish were collected hence species richness = 0 (zero). To change these values to zero, we can use ifelse() like we did above, except we change the NA to zero. Examine and run the code below. *Did it take care of the “missing” richness data?*

```
# replace NA with zero
for.correl$detect<-
ifelse(is.na(for.correl$detect),0,for.correl$detect)
# print it out
for.correl
```

Correlation

To calculate a correlation between habitat variables and species richness, we can use the cor() function. The cor function uses 2 or more vectors (i.e., matrices) and calculated the correlations among these objects. For example, the code below specifies a correlation between detect (richness) and sample unit length, both in the for.correl data frame. Examine the code and run.

```
cor(for.correl$detect,for.correl$Length)
```

What happened after you submitted the code? You got an NA, didn't you? Why do you think this happened? Well, we got NA when we tried to calculate a mean with missing data another reason for NA could be that we were trying to calculate a correlation using a factor or character variable. So let's first use summary to see what's in for.correl.

```
# summarize for.correl
summary(for.correl)
```

Submitting the above should give you:

```
                    Stream          Date       Habitat        detect           Length          Width            Depth      
 Howards Branch          :9   22-Jul-09:16   Pool  :22   Min.   :0.000   Min.   : 3.00   Min.   :1.200   Min.   :0.06000  
 L Ball Crk              :9   23-Jul-09:29   Riffle:18   1st Qu.:0.000   1st Qu.: 8.55   1st Qu.:1.600   1st Qu.:0.09525  
 L Drymans Fork          :9   5-Aug-09 : 6   Run   :11   Median :2.000   Median :11.30   Median :2.265   Median :0.13000  
 Sheppard Crk            :6                              Mean   :2.078   Mean   :11.48   Mean   :2.903   Mean   :0.16594  
 WTSD 7                  :6                              3rd Qu.:4.000   3rd Qu.:14.00   3rd Qu.:4.000   3rd Qu.:0.18500  
 High White Crk (WTSD 14):5                              Max.   :8.000   Max.   :23.40   Max.   :6.800   Max.   :0.60000  
 (Other)                 :7                                              NA's   :3       NA's   :3       NA's   :3        
    Velocity          Cobble          Gravel     
 Min.   :0.0300   Min.   : 0.00   Min.   : 0.00  
 1st Qu.:0.1325   1st Qu.:15.00   1st Qu.: 5.00  
 Median :0.2140   Median :20.00   Median :20.00  
 Mean   :0.2381   Mean   :25.71   Mean   :18.47  
 3rd Qu.:0.3000   3rd Qu.:30.00   3rd Qu.:30.00  
 Max.   :0.6600   Max.   :90.00   Max.   :50.00  
 NA's   :1        NA's   :2       NA's   :2      
```

Looks like detect and Length are numeric (summary calculated means and percentiles), however Length has 3 NA’s. This is probably the reason we got the NA when we tried to calculate a correlation. You may be tempted to use na.rm = T DO NOT. First check the syntax of cor() using the help() function.

```
help(cor)
```

The help file indicates that we handle NA with the “use =” option. There are several options available for use, here we will specify use = "pairwise.complete.obs". Below is the code for calculating Pearson correlations between richness (detect) and sample unit length and among data in columns 4 to 10: detect, Length, Width, Depth, Velocity, Cobble, and Gravel.

```
cor(for.correl$detect,for.correl$Length, use =
"pairwise.complete.obs")

cor(for.correl[,4:10], use = "pairwise.complete.obs")
```

BASIC R GRAPHICS

Note--- We will not be making fancy "infovis" type graphics... Just plain ole' boring graphics you will see in reports and publications.

Read in the data and summary

Set the working directory

```
# setwd('.../your directory here/...')
dat <- read.csv("dat.csv")
```

What the data looks like.

```
head(dat)
##   X ID year length weight stage
## 1 1  1 2009  701.0  30.30 Adult
## 2 2  2 2012  593.9  52.78 Adult
## 3 3  3 2012  729.5 101.98 Adult
## 4 4  4 2010  563.2  65.93 Adult
## 5 5  5 2011  231.1  10.94   Juv
## 6 6  6 2012  723.0  77.13 Adult
str(dat)
## 'data.frame':    1000 obs. of  6 variables:
##  $ X     : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ ID    : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ year  : int  2009 2012 2012 2010 2011 2012 2011 2010 2012 2009
...
##  $ length: num  701 594 729 563 231 ...
##  $ weight: num  30.3 52.8 102 65.9 10.9 ...
##  $ stage : Factor w/ 2 levels "Adult","Juv": 1 1 1 1 2 1 1 1 1 1
...
summary(dat)
##        X                ID              year          length
##  Min.   :   1   Min.   :   1   Min.   :2009   Min.   :200
##  1st Qu.: 251   1st Qu.: 251   1st Qu.:2009   1st Qu.:338
##  Median : 500   Median : 500   Median :2010   Median :489
##  Mean   : 500   Mean   : 500   Mean   :2010   Mean   :493
##  3rd Qu.: 750   3rd Qu.: 750   3rd Qu.:2011   3rd Qu.:644
##  Max.   :1000   Max.   :1000   Max.   :2012   Max.   :800
##      weight          stage
##  Min.   :  0.4   Adult:857
##  1st Qu.: 10.2   Juv  :143
##  Median : 31.3
##  Mean   : 76.6
##  3rd Qu.: 90.8
##  Max.   :700.5
```

```
names(dat)
## [1] "X"      "ID"     "year"   "length" "weight" "stage"
```

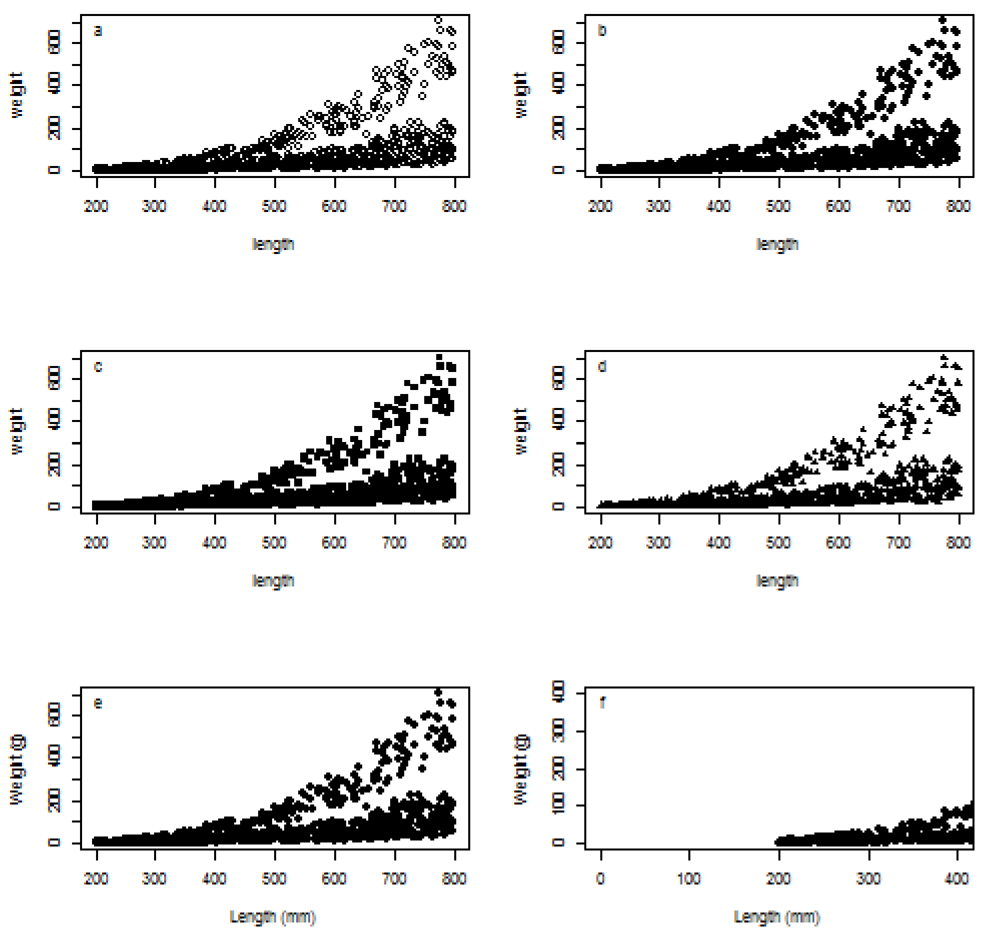
Just how much data are we dealing with?

```
dim(dat)
## [1] 1000    6
nrow(dat)
## [1] 1000
ncol(dat)
## [1] 6
```

Ok all looks good. Let’s start plotting.

Scatter plots

The following code (below code) will make the figure below.



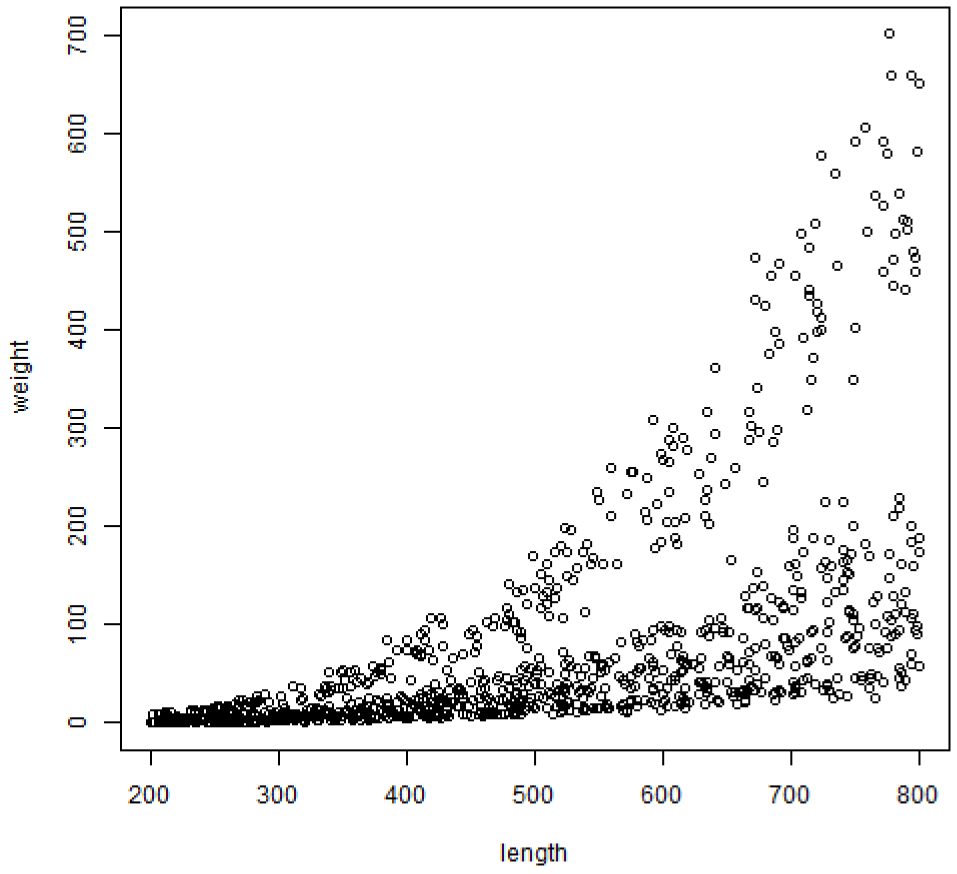
Making a default scatterplot

Let's start with panel a. This is your basic plot of length versus weight. The default R code to do this is

```
plot(weight ~ length, dat)
```

Note the formula notation which is y variable ~ x variable you could as well, but I prefer formula notation.

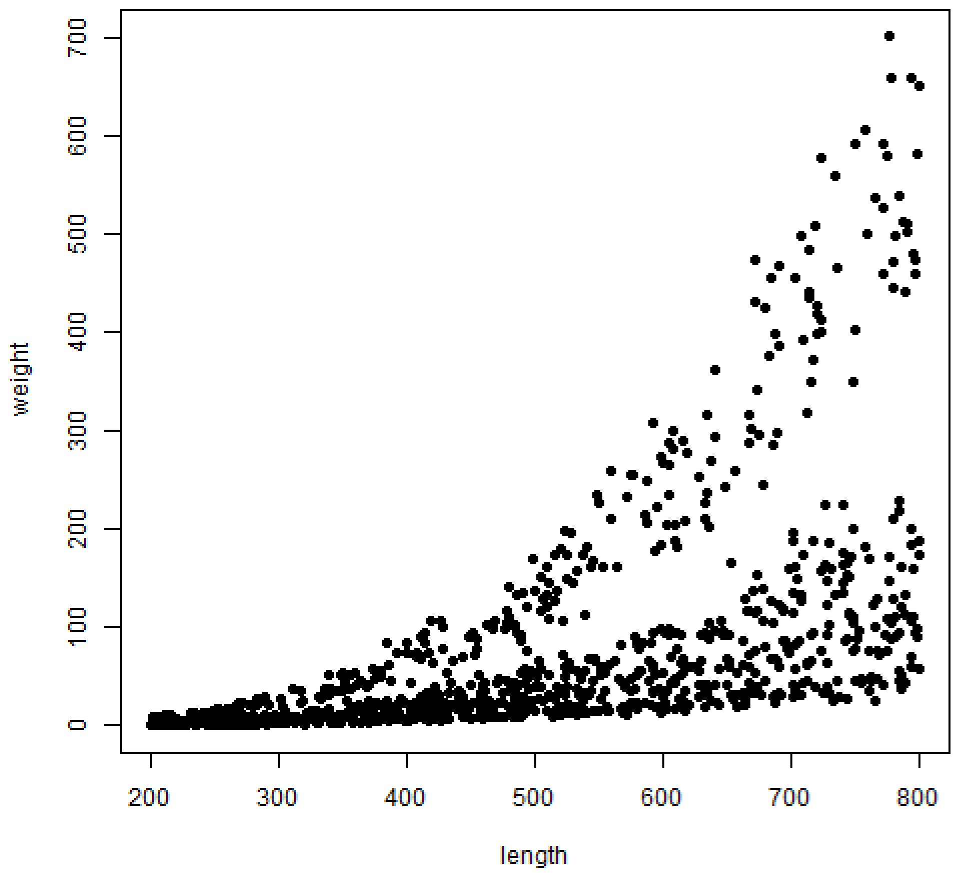
```
plot(dat$length,dat$weight)
```



Changing the plotting character

We can do this by changing the pch and we can reproduce panel b from Figure 1. See the material at the end of the lab for the pch values that correspond to the plotting values.

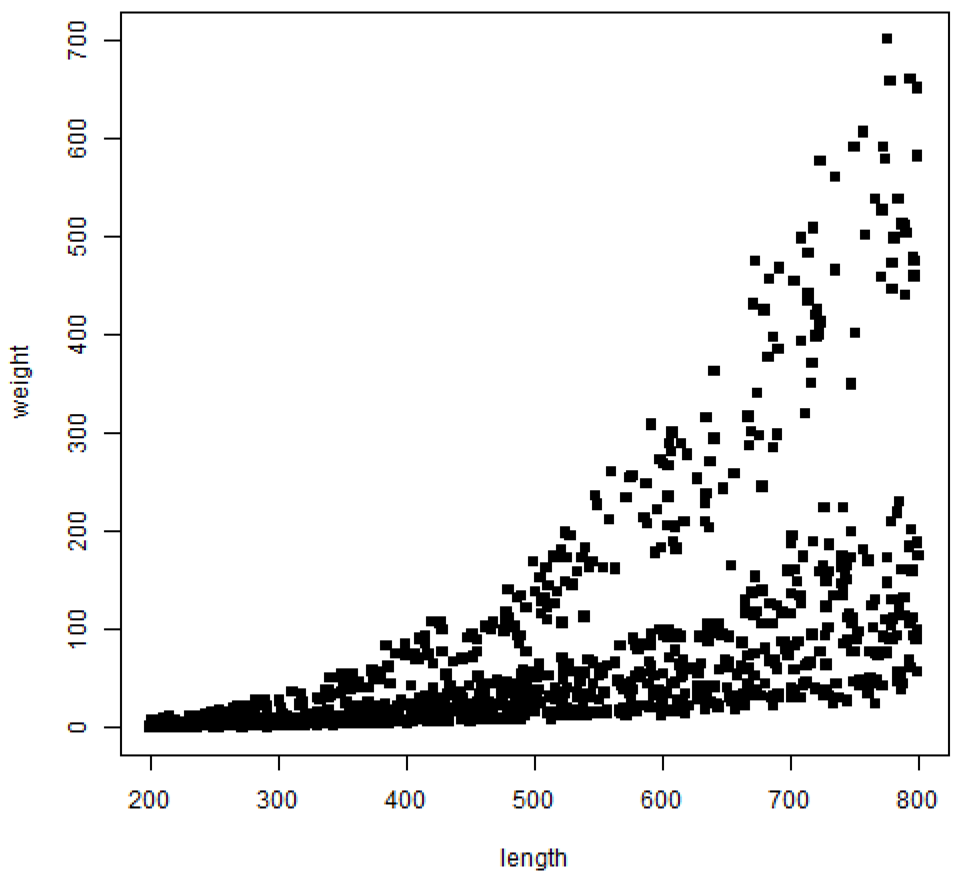
```
plot(weight ~ length, dat, pch = 19)  # filled circles!
```



The above plot replicates panel B in top figure.

What about if you want filled in squares? Easy peasy, just use pch again but this time with a value of 15.

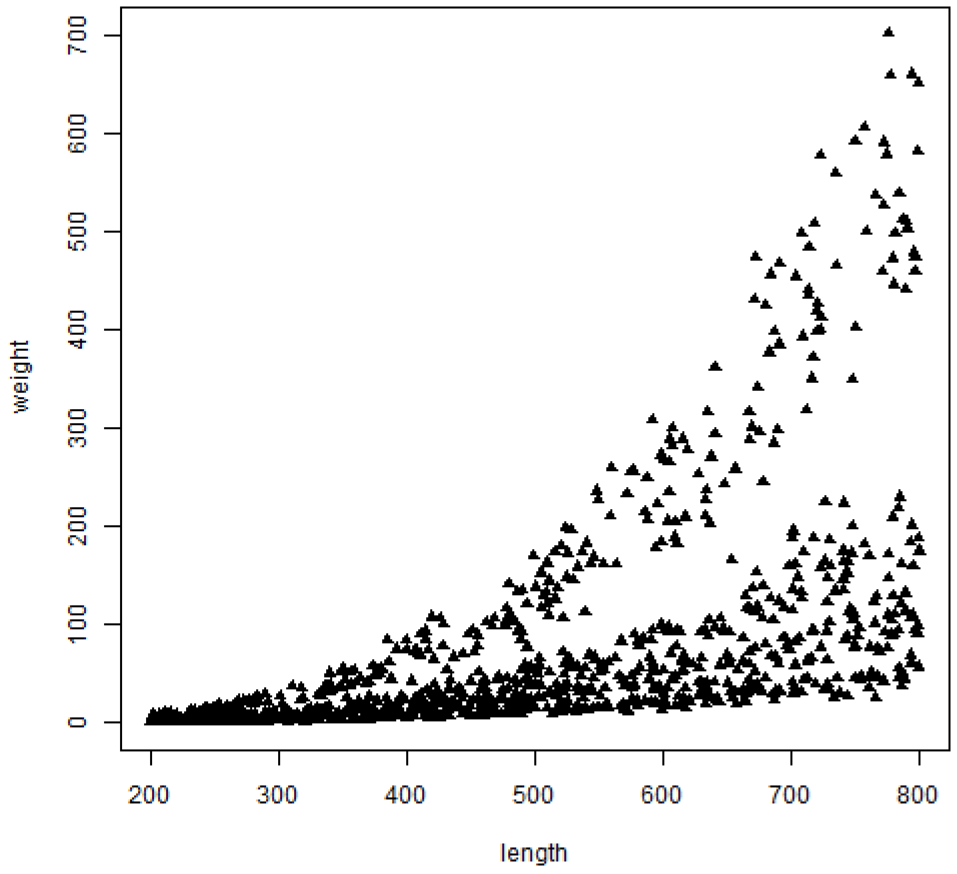
```
plot(weight ~ length, dat, pch = 15)  # filled squares
```



The above plot replicates panel C in top figure.

Wildcard! What about filled in triangles? Again easy peasy, just use pch again but this time with a value of 17.

```
plot(weight ~ length, dat, pch = 17)  # filled triangles
```

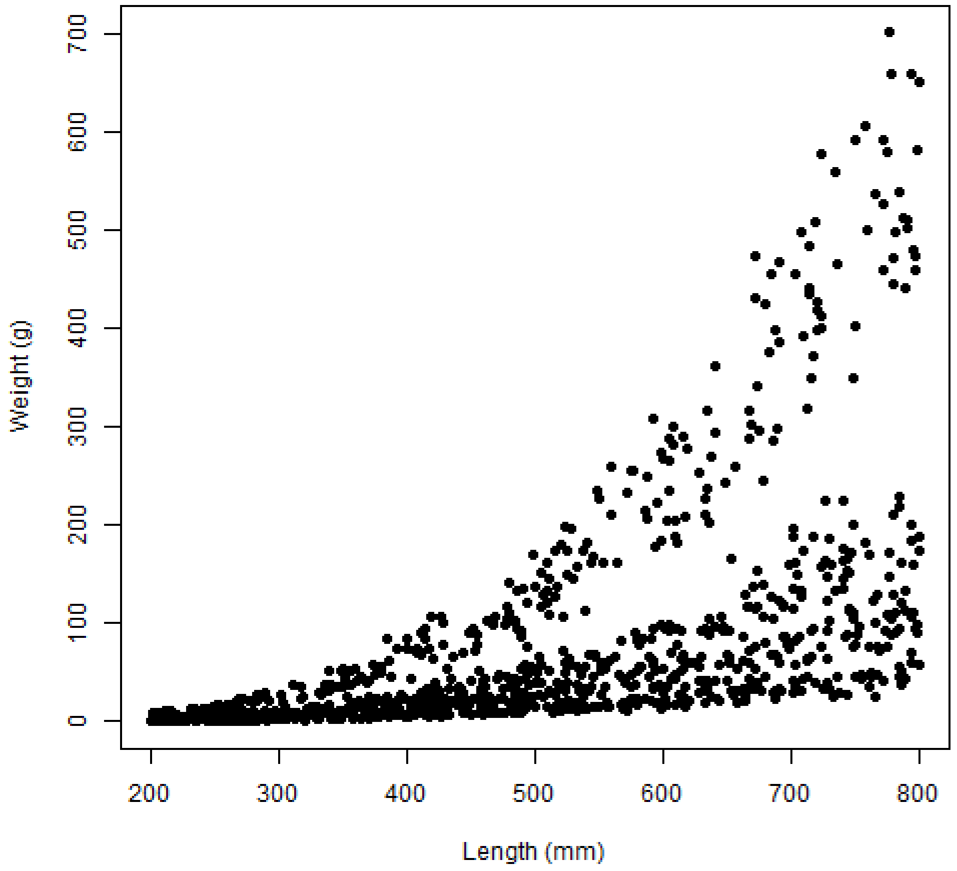


The above plot replicates panel D in top figure.

Adding custom x- and y-axis labels

That was a fun exercise, but we come full circle and have decided that filled in circles will work best (full circle… get it?). But, being good researchers we want to change the x and y-axis labels to incorporate the units. We need to do this because R by default uses the name of the column. So we could rename the column in the data.frame, but there has to be a better way. All we need to add a custom axis label by adding a xlab and ylab argument. The following code demonstrates this.

```
plot(weight ~ length, dat, pch = 19, xlab = "Length (mm)", ylab
= "Weight (g)")
```

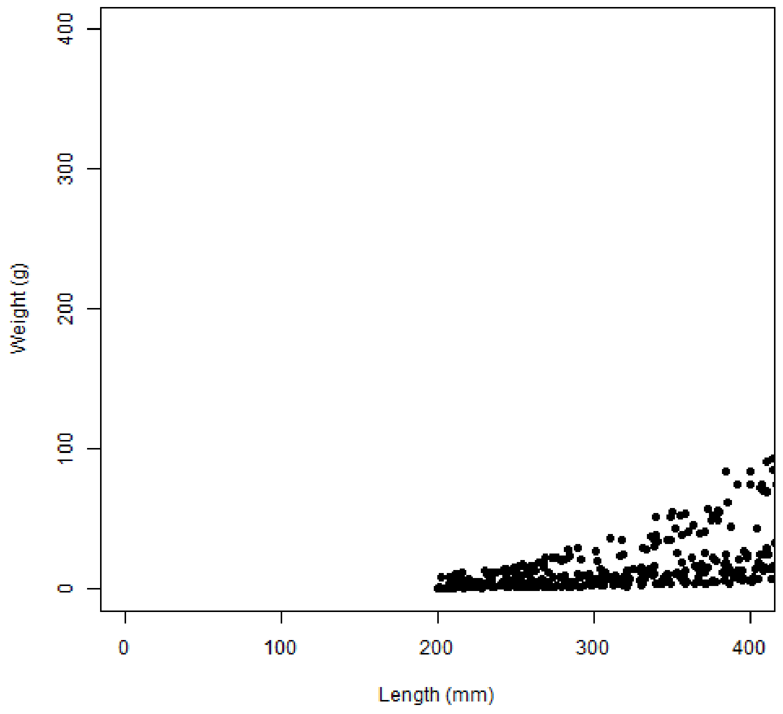


The figure above replicates panel E in top figure.

Changing the plotting x- and y-axis limits

We can also use the xlim and ylim argument to change the graphic area plotted. The code below produces a figure that limits the x- and y-axis to be between 0 and 400.

```
plot(weight ~ length, dat, pch = 19, xlab = "Length (mm)", ylab
= "Weight (g)", xlim = c(0, 400), ylim = c(0, 400))
```



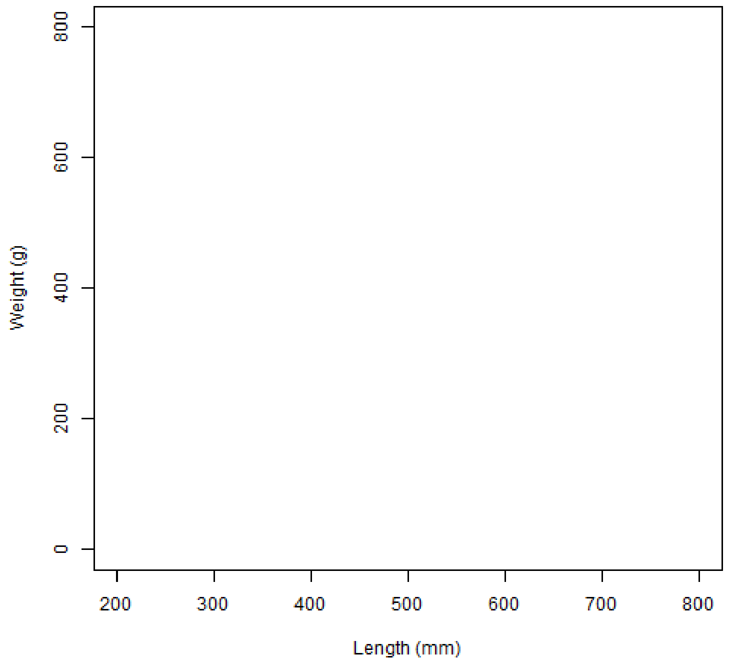
The figure above replicates panel F in Top figure.

Plotting groups: layers on a blank canvas

Wow! The data looks like it has some structure in it that we could illustrate by grouping some of the data and plotting it using different colors for hears in the data.

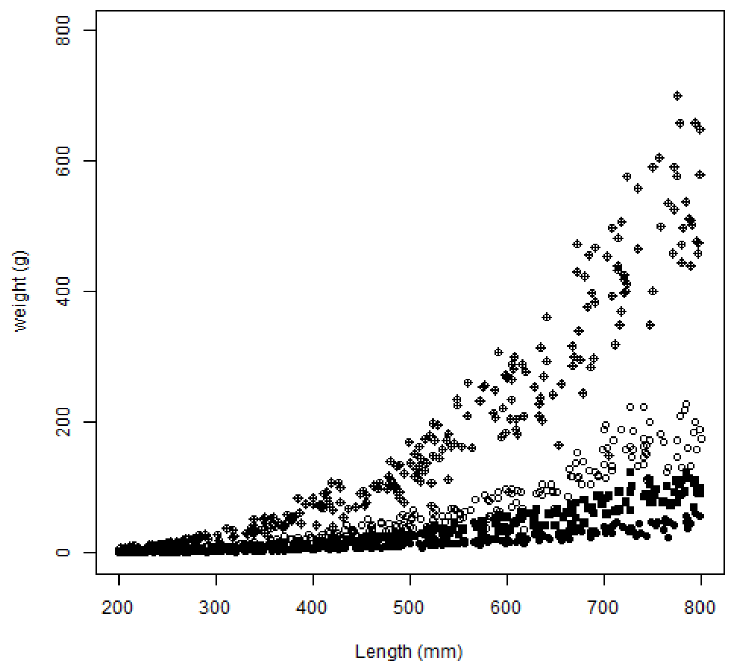
First we are going to start with a 'blank canvas to plot on by specifying type='n' which will give us the plot below.

```
plot(weight ~ length, dat, xlab = "Length (mm)", ylab = "Weight
(g)", xlim = c(200,800), ylim = c(0, 800), type = "n")
```



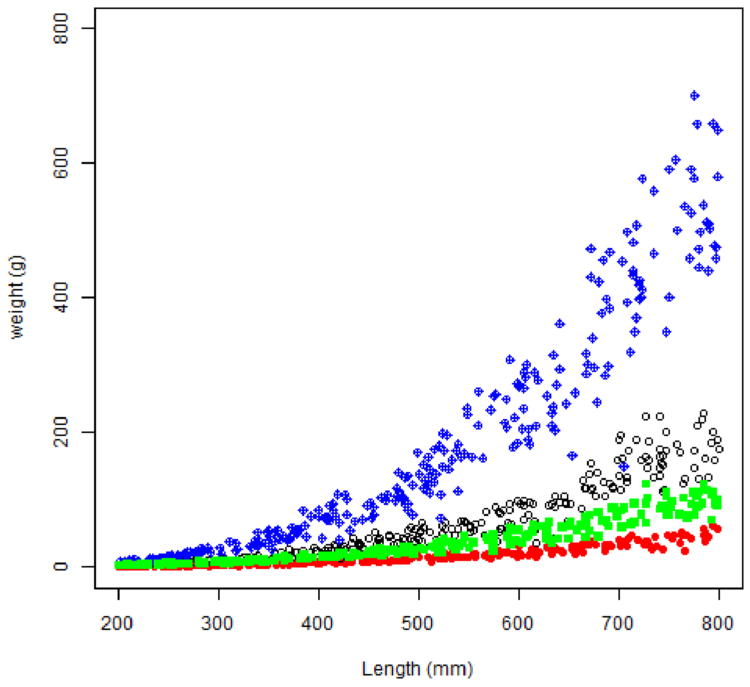
Now we can add things to the plot. This is accomplished using the points() function. Let’s add some data for 2009. The points() function has a subset argument that we can specify a column in the data and some sort of subsetting argument

```
plot(weight ~ length, dat, xlab = "Length (mm)", ylab = "weight
(g)", xlim = c(200,
    800), ylim = c(0, 800), type = "n")
points(weight ~ length, dat, subset = year == 2009, pch = 19)
points(weight ~ length, dat, subset = year == 2010, pch = 1)
points(weight ~ length, dat, subset = year == 2011, pch = 10)
points(weight ~ length, dat, subset = year == 2012, pch = 15)
```



But the above figure is a bit difficult to differentiate, I certainly wouldn't use in a presentation, how about we jazz it up with some color by
adding a col argument.

```
plot(weight ~ length, dat, xlab = "Length (mm)", ylab = "weight
(g)", xlim = c(200,800), ylim = c(0, 800), type = "n")
points(weight ~ length, dat, subset = year == 2009, pch = 19,
col = "red")  # add data for 2009
points(weight ~ length, dat, subset = year == 2010, pch = 1, col
= "black")  # add data for 2010
points(weight ~ length, dat, subset = year == 2011, pch = 10,
col = "blue")  # add data for 2011
points(weight ~ length, dat, subset = year == 2012, pch = 15,
col = "green")  # add data for 2012
```

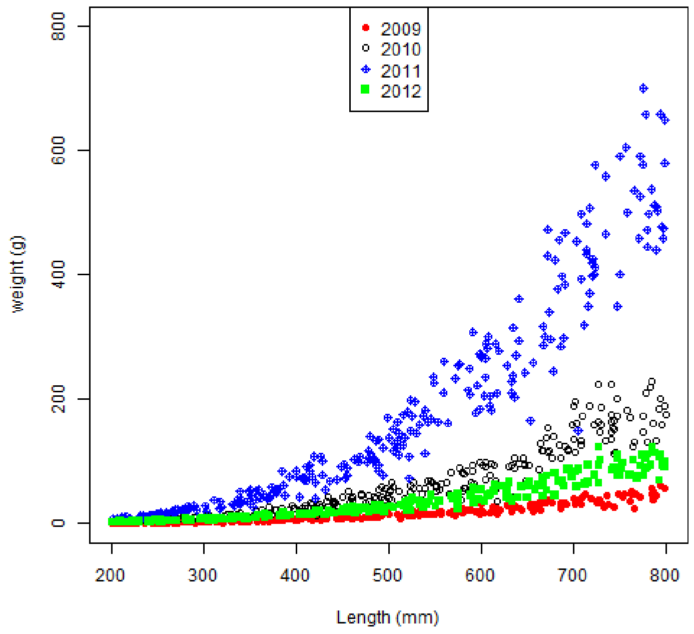


Wow that really pops! Now how do we know what means what? We need a legend, not a mythical story but a way to make the colors and symbols mean something.

Adding a legend to a plot

The bare minimum that we need to specify to the legend() function is where it should be located, what is the legend, what are the symbols used, and what color are those symbols. Be careful here, order is important! Let's recreate the plot above and add a legend.

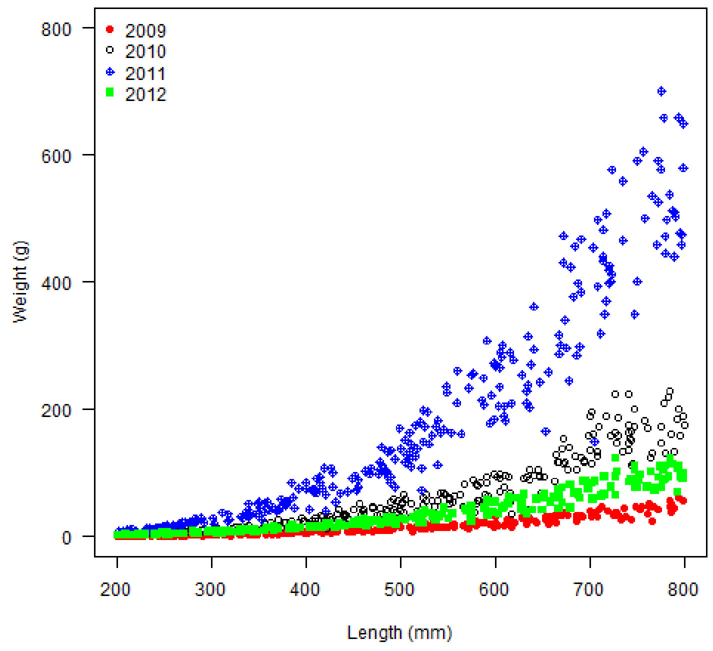
```
plot(weight ~ length, dat, xlab = "Length (mm)", ylab = "weight
(g)", xlim = c(200,800), ylim = c(0, 800), type = "n")
points(weight ~ length, dat, subset = year == 2009, pch = 19,
col = "red")  # add data for 2009
points(weight ~ length, dat, subset = year == 2010, pch = 1, col
= "black")  # add data for 2010
points(weight ~ length, dat, subset = year == 2011, pch = 10,
col = "blue")  # add data for 2011
points(weight ~ length, dat, subset = year == 2012, pch = 15,
col = "green")  # add data for 2012
## Adding a default legend
legend("top", legend = c("2009", "2010", "2011", "2012"), pch =
c(19, 1, 10, 15), col = c("red", "black", "blue", "green"))
```



The use of 'top' as the first argument tells R where to plot the legend. The rest gives the text legend, symbol and color. The location argument can be: “topleft, top, topright, left, right, bottomleft, bottom, bottomright.

The code below makes the same previous plot but with the legend in the top left hand corner.

```
plot(weight ~ length, dat, xlab = "Length (mm)", ylab = "Weight
(g)", xlim = c(200, 800), ylim = c(0, 800), type = "n")
points(weight ~ length, dat, subset = year == 2009, pch = 19,
col = "red")  # add data for 2009
points(weight ~ length, dat, subset = year == 2010, pch = 1, col
= "black")  # add data for 2010
points(weight ~ length, dat, subset = year == 2011, pch = 10,
col = "blue")  # add data for 2011
points(weight ~ length, dat, subset = year == 2012, pch = 15,
col = "green")  # add data for 2012
## Adding a default legend
legend("topleft", legend = c("2009", "2010", "2011", "2012"),
pch = c(19, 1,10, 15), col = c("red", "black", "blue", "green"))
```



Some personal preferences

I prefer my y-axis labels to be parallel to the x-axis which can be specified by the argument las=1. I also prefer my legends to not have a box which is taken care of with the argument bty='n'

```
plot(weight ~ length, dat, xlab = "Length (mm)", ylab = "Weight
(g)", xlim = c(200,800), ylim = c(0, 800), type = "n", las = 1)
points(weight ~ length, dat, subset = year == 2009, pch = 19,
col = "red")  # add data for 2009
points(weight ~ length, dat, subset = year == 2010, pch = 1, col
= "black")  # add data for 2010
points(weight ~ length, dat, subset = year == 2011, pch = 10,
col = "blue")  # add data for 2011
points(weight ~ length, dat, subset = year == 2012, pch = 15,
col = "green")  # add data for 2012
## Adding a default legend
```

```
legend("topleft", legend = c("2009", "2010", "2011", "2012"),
pch = c(19, 1, 10, 15), col = c("red", "black", "blue",
"green"), bty = "n")
```

Line plots

Most of what we have learned for scatterplots applies for line plots! Lest plot the mean wieght by year for the data as a line plot. First we need to aggregate the dataset in the code below and make a new dataframe lindat .

Aggregating the dataset

```
lindat <- aggregate(weight ~ year, dat, mean)
lindat$length <- aggregate(length ~ year, dat, mean)$length
lindat$n <- aggregate(weight ~ year, dat, length)$weight
lindat$var <- aggregate(weight ~ year, dat, var)$weight
lindat$lci <- lindat$weight - 1.96 *
sqrt(lindat$var)/sqrt(lindat$n)
lindat$uci <- lindat$weight + 1.96 *
sqrt(lindat$var)/sqrt(lindat$n)
```

The code we will be working through will step through using lindat created by the code above to plot panels A-D below.

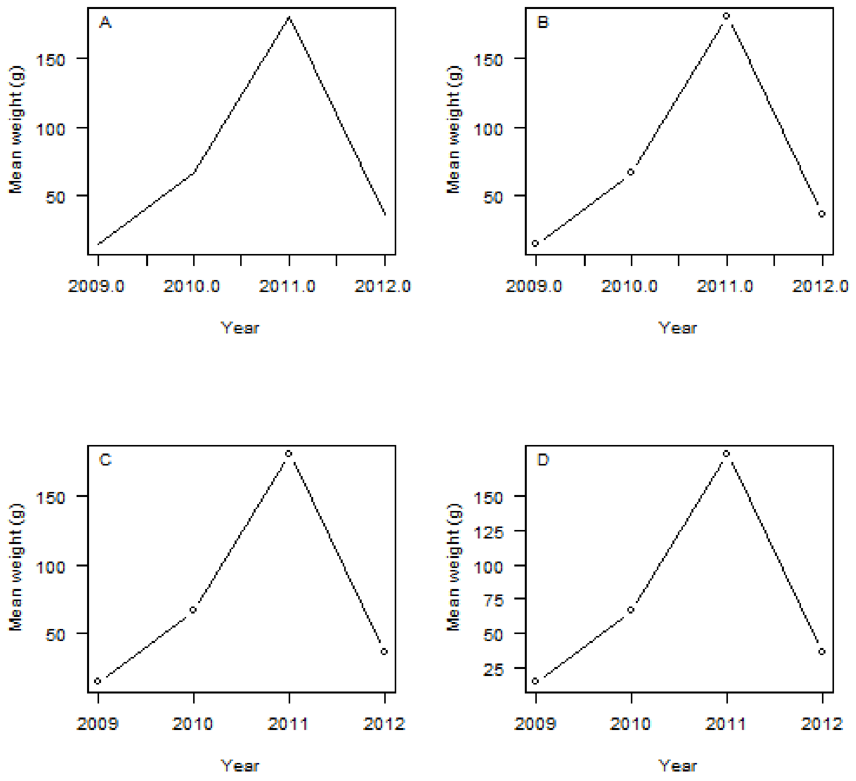
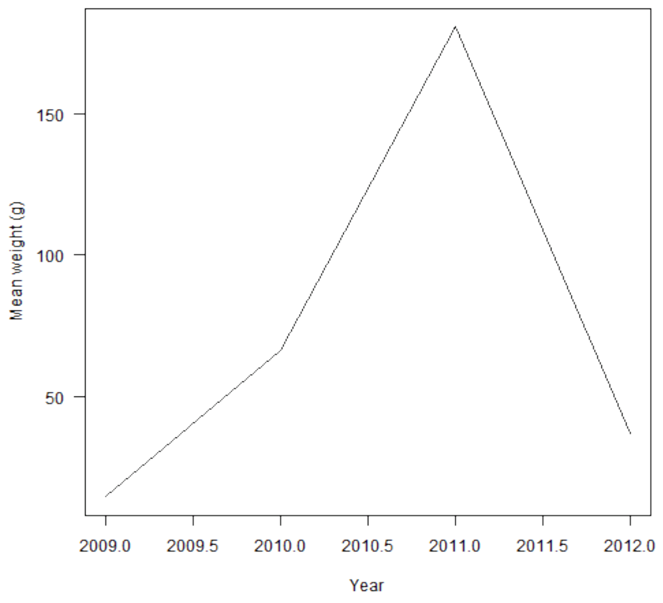


Figure 2.—The code below iteratively works through plots A-D above.

Default line plots

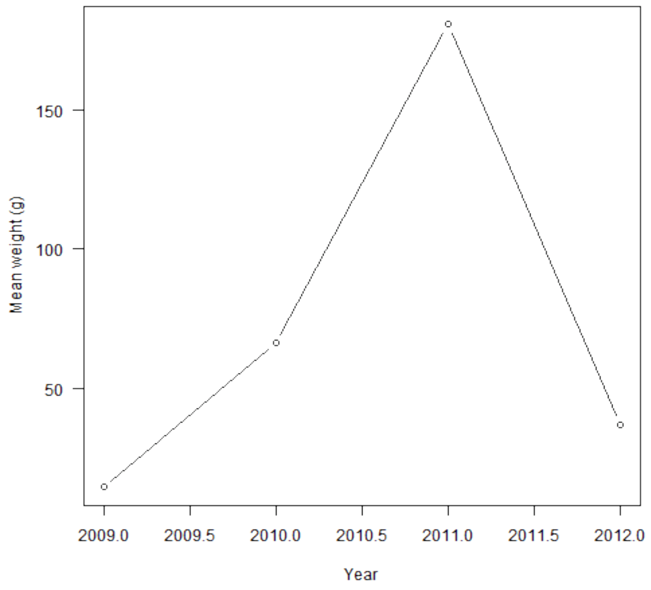
Line plots are done by specifying the argument type='l'. By default is it type='p' which returns a scatter plot as we just worked through. Lets plot mean weight by year using the code below

```
plot(weight ~ year, lindat, type = "l", las = 1, xlab = "Year",
ylab = "Mean weight (g)")
```



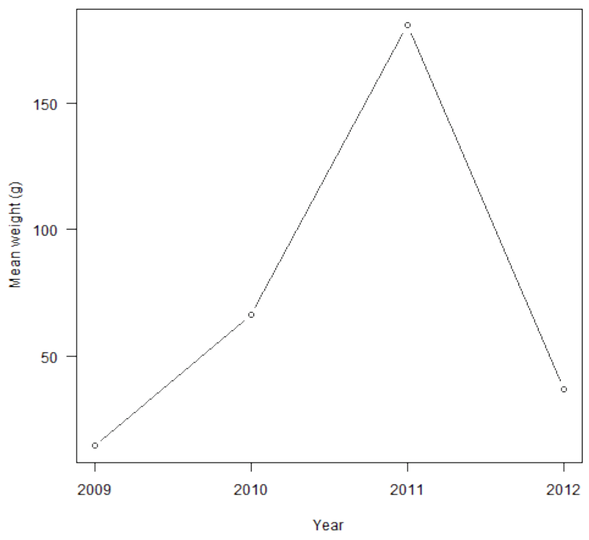
Well that plot is a bit misleading, we don't actually have data for all points on the line but it illustrates trends really well. How about a mix of points for data and a line to illustrate trend. This can be done using the argument type='b' where the b is short for both as in both points and lines.

```
plot(weight ~ year, lindat, type = "b", las = 1, xlab = "Year",
ylab = "Mean weight (g)")
```



OK now I am just nit picking this thing to death, but that is what we do right? What is year 2010.5, especially if we don't have any data? We can deal with this by turning off the x-axis in the plot using the xaxt='n' argument and specifying our own custom one using the axis() function.

```
# fix the x-axis (xaxt='n')
plot(weight ~ year, lindat, type = "b", las = 1, xlab = "Year",
ylab = "Mean weight (g)", xaxt = "n")
axis(side = 1, at = c(2009, 2010, 2011, 2012), labels =
c("2009", "2010", "2011", "2012"))
```



In the axis() function the arguments specify the following:

side: what side to add the axis to (1=bottom, 2=left, 3=top, 4=left)

at: where to put tick marks

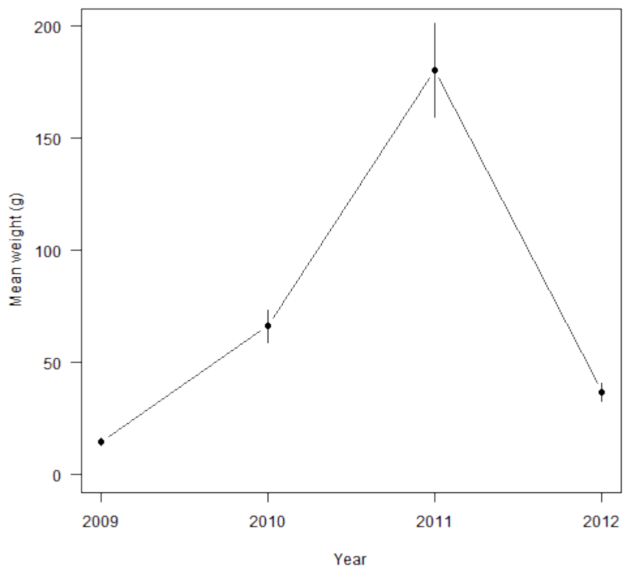
labels: what to label those ticks marks

Adding confidence intervals

Lines illustrating confidence intervals (or any sort of line you may want) can be added using the segments() function. This function requires two points, a start (x0, y0) and an end (x1, y1). To add lines representing a confidence interval all you need to do is specify the x location and then the lower and upper CI.

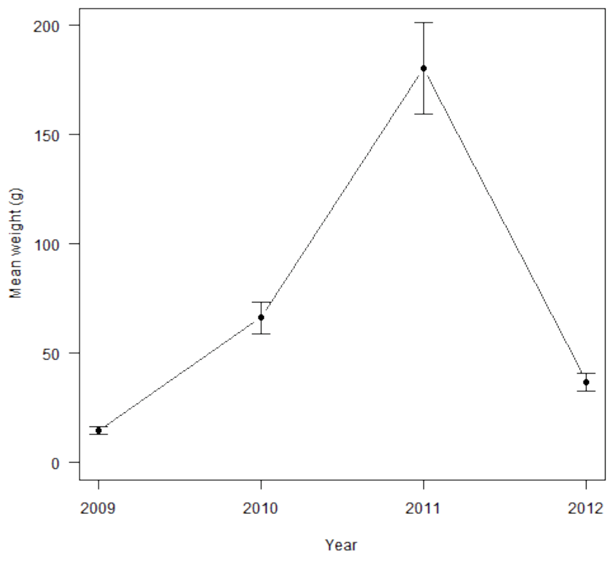
Figure 3.—The code below was used to construct this figure demonstrating how confidence intervals can be illustrated on plots using the segments() and arrows() functions

```
plot(weight ~ year, lindat, type = "b", las = 1, xlab = "Year",
ylab = "Mean weight (g)", xaxt = "n", ylim = c(0, 200), pch =
19)
axis(side = 1, at = c(2009, 2010, 2011, 2012), labels =
c("2009", "2010", "2011", "2012"))
segments(x0 = lindat$year, y0 = lindat$lci, x1 = lindat$year, y1
= lindat$uci)
```



We can get more traditional 'whiskers' using the arrows() function and specifying that the arrow angle using the argument angle=90. The length of the end arrow is

```
# whiskers
plot(weight ~ year, lindat, type = "b", las = 1, xlab = "Year",
ylab = "Mean weight (g)", xaxt = "n", ylim = c(0, 200), pch =
19)
axis(side = 1, at = c(2009, 2010, 2011, 2012), labels =
c("2009", "2010", "2011", "2012"))
arrows(x0 = lindat$year, y0 = lindat$lci, x1 = lindat$year, y1 =
lindat$uci, angle = 90, length = 0.1, code = 3)
```

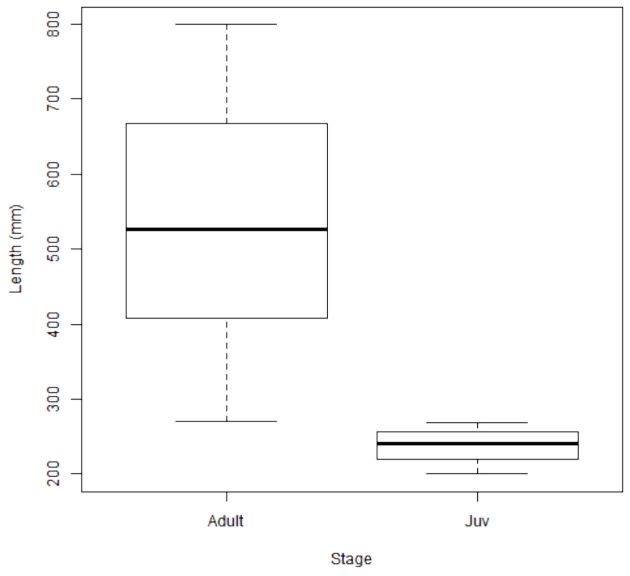


In the arrows() function the arguments specify the following:
angle: angle of the arrow…90 are 90 degree angles
length: how long should those arrows be?
code: what kind of arrow (1=arrow head at beginning, 2 = arrowhead at end, 3 = arrowhead at beginning and end)

Boxplots

Boxplots are created by the boxplot() function. R takes a continuous variable and then factors the conditioning variable if it is not already. Let look at length by stage in our data.

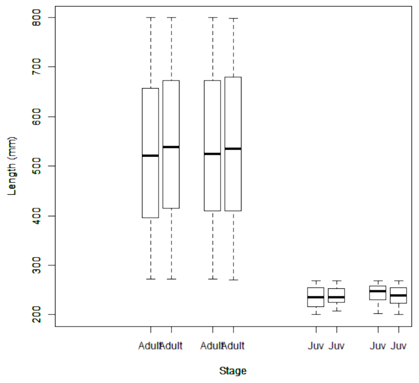
```
boxplot(length ~ stage, dat, xlab = "Stage", ylab = "Length
(mm)")
```



As before we had some strong variation among years, it might be good to look at that! Let's plot boxplots for each stage and year combination. This is done by subsetting and adding the plots. But we have to be careful to plot them at the right location on the x-axis. Basically we have to shift them left or right from the plotting location. This can be done using the "at" argument. We also have to resize the boxes so that width is not so large that the boxes overlap.

```
boxplot(length ~ stage, dat, xlab = "Stage", ylab = "Length
(mm)", subset = year == 2009, at = c(1:2) - 0.25, boxwex = 0.1)
```

```
boxplot(length ~ stage, dat, xlab = "Stage", ylab = "Length
(mm)", subset = year == 2010, add = TRUE, at = c(1:2) - 0.125,
boxwex = 0.1)
boxplot(length ~ stage, dat, xlab = "Stage", ylab = "Length
(mm)", subset = year == 2011, add = TRUE, at = c(1:2) + 0.125,
boxwex = 0.1)
boxplot(length ~ stage, dat, xlab = "Stage", ylab = "Length
(mm)", subset = year == 2012, add = TRUE, at = c(1:2) + 0.25,
boxwex = 0.1)
```



In the boxplot() function the arguments specify the following:

at: where the groups should be plotted

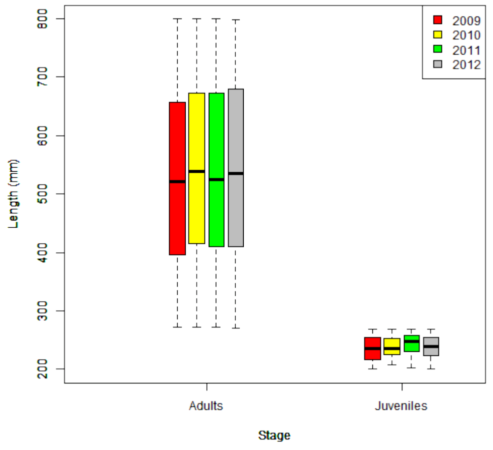
”boxwex: width of the box

add=TRUE: should the boxplot be added to the existing plot?

Well that was fun but that x-axis is atrocious. We can fix this with xaxt as we did before to suppress plotting of the x-axis.

```
boxplot(length ~ stage, dat, xlab = "Stage", ylab = "Length
(mm)", subset = year == 2009, at = c(1:2) - 0.15, boxwex = 0.08,
xaxt = "n", col = "red")
boxplot(length ~ stage, dat, xlab = "Stage", ylab = "Length
(mm)", subset = year == 2010, add = TRUE, at = c(1:2) - 0.05,
boxwex = 0.08, xaxt = "n", col = "yellow")
```

```
boxplot(length ~ stage, dat, xlab = "Stage", ylab = "Length
(mm)", subset = year == 2011, add = TRUE, at = c(1:2) + 0.05,
boxwex = 0.08, xaxt = "n", col = "green")
boxplot(length ~ stage, dat, xlab = "Stage", ylab = "Length
(mm)", subset = year == 2012, add = TRUE, at = c(1:2) + 0.15,
boxwex = 0.08, xaxt = "n", col = "grey")
axis(side = 1, at = c(1, 2), labels = c("Adults","Juveniles"))
legend("topright", legend = c(2009:2012), fill = c("red",
"yellow", "green", "grey"))
```



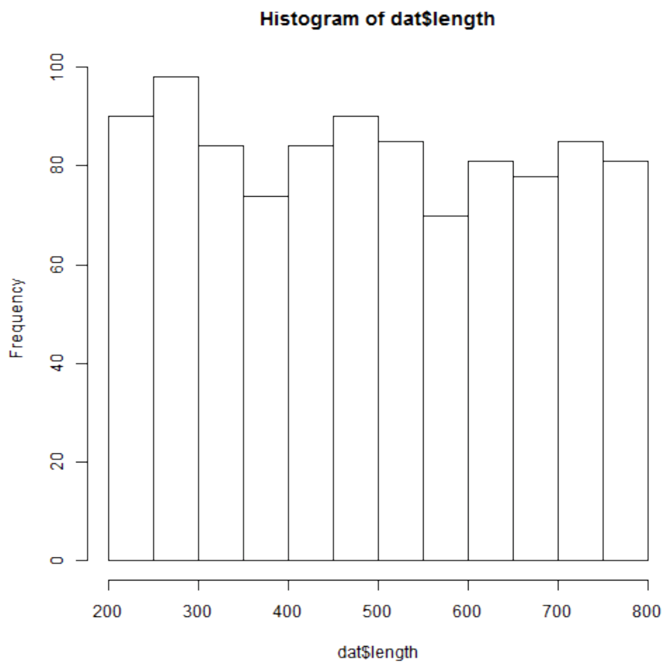
In the legend() function the arguments specify the following:

fill: what color to fill boxes in the legend

Histograms

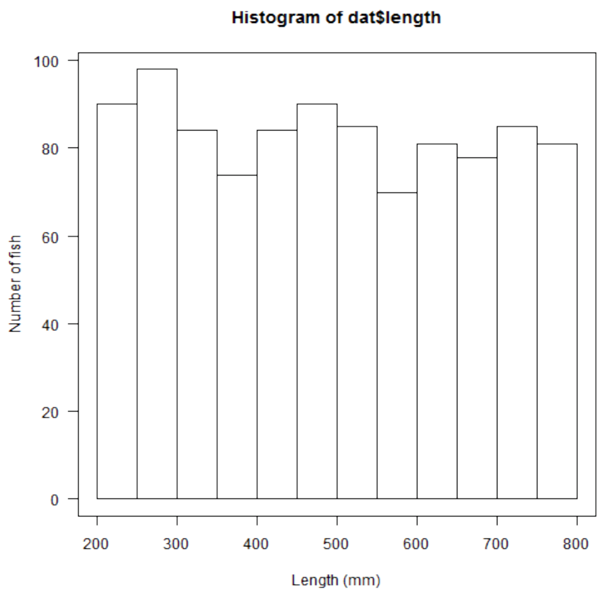
Histograms of a vector of data are easily made using the histogram() function. All you have to do is feed it a vector of data and it will bin it for you and you are off and running! But the default is not as pretty as it could be.

```
hist(dat$length)
```



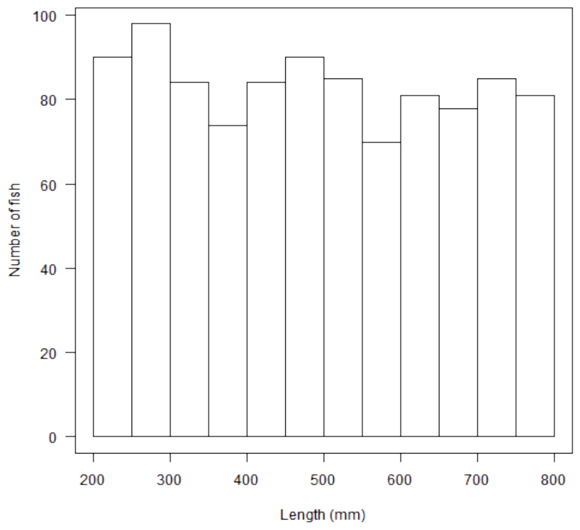
Let’s clean it up some by adding a box and add some labels.

```
hist(dat$length, xlab = "Length (mm)", ylab = "Number of fish",
las = 1)
box()
```



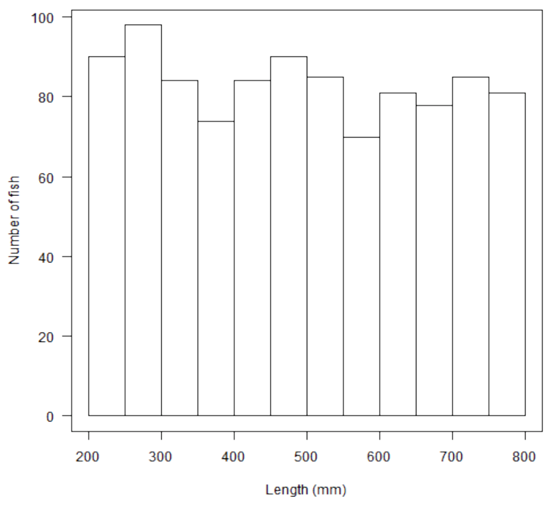
Uggg that title is atrocious, let’s just get rid of it all together. We can do this by the argument main="" or main=NULL.

```
hist(dat$length, xlab = "Length (mm)", ylab = "Number of fish",
las = 1, main = "")
box()
```



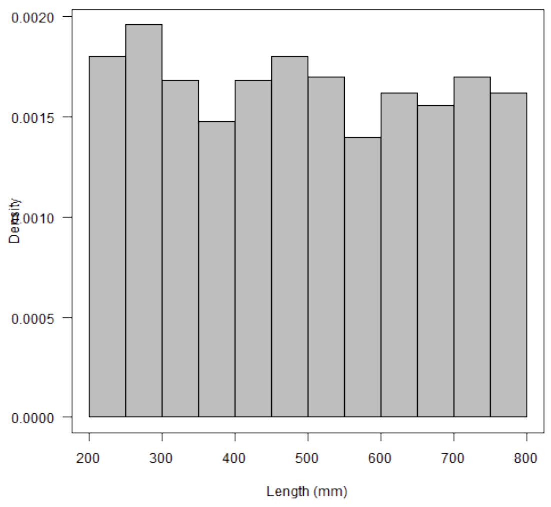
This code produces the same thing as the above code.

```
hist(dat$length, xlab = "Length (mm)", ylab = "Number of fish",
las = 1, main = NULL)
box()
```



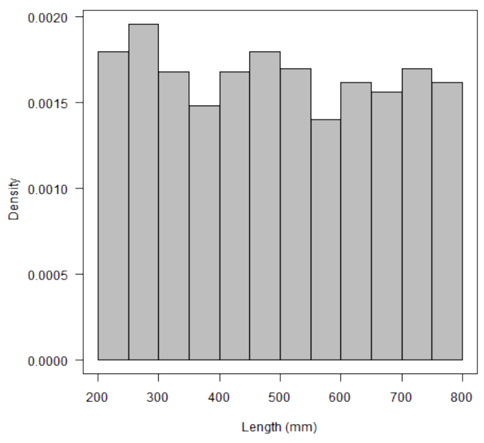
Let’s add some color to up the ink ratio. But now we are going plot the density and not the frequency (the default is to plot the frequency).

```
hist(dat$length, xlab = "Length (mm)", ylab = "Density", las =
1, main = "", freq = FALSE, col = "grey")
box()
```



Whoa Nelly… Now I am having a hard time reading my y-axis label. We really need to customize it. We can do this using the mtext() function to plot text in the margins. Let’s give it a shot. One thing we need to do is use the par() function to change the margins a bit to give us some room to work.

```
par(oma = c(1, 2, 1, 1))  # add an outer margin to plot in
hist(dat$length, xlab = "Length (mm)", ylab = "", las = 1, main
= "", freq = FALSE, col = "grey")
box()
mtext(side = 2, "Density", outer = TRUE, line = 0)
```



That is much better.

Code breakdown:

oma: specifies the margin widths (in lines ~12pt), you need to enter 4 values going from bottom, left, top, right… this mathces how side is used in the axis() function

side=2: tells R to plot the text in the left margin

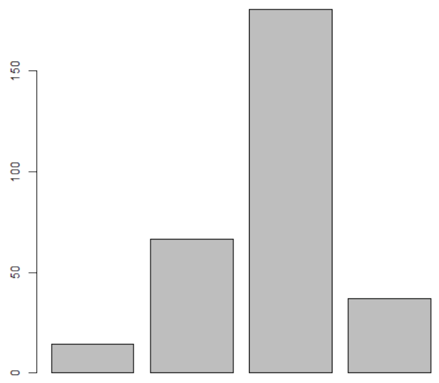
outer=TRUE: tells R to plot in the outer margin

line=0: plots the text on line 0

Bar plots

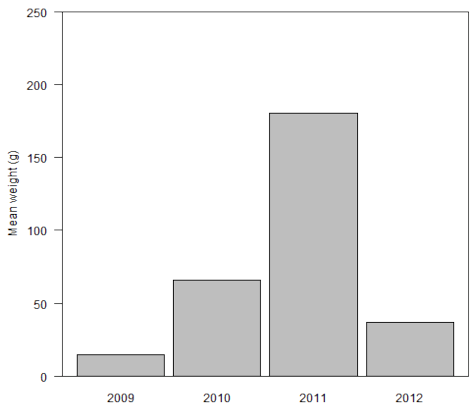
No this is not hanging out at Bombs making plots… Lets go ahead and make a default barplot using our summary data lindat with the mean weights.

```
barplot(lindat$weight)
```



The plot above is almost as satisfying as plotting in Excel. Lets clean it up some and make is presentable by adding a box around it and some labels.

```
barplot(lindat$weight, names = lindat$year, las = 1, yaxt = "n",
ylim = c(0,
    250), ylab = "Mean weight (g)", width = 0.9, space = 0.1)
axis(side = 2, at = seq(0, 250, 50), labels = TRUE, las = 1)
box()
```



Code breakdown:

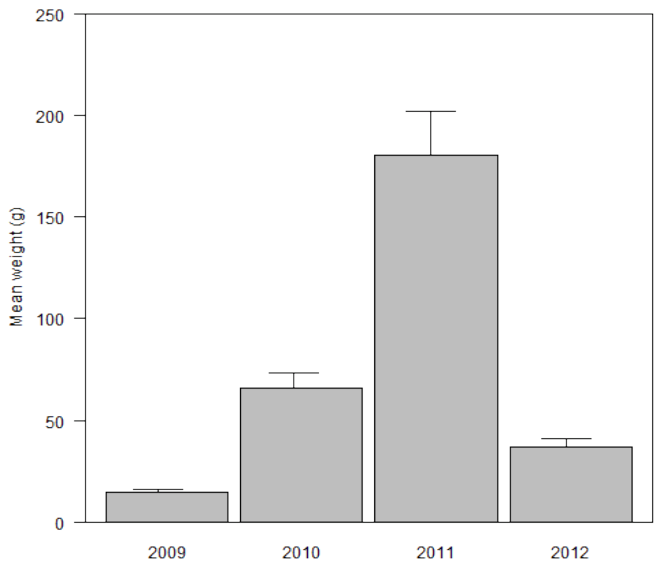
width: how wide to make the bars

space: how much space between bars

Barplots with error bars (aka dynamite plots)

Lets tweak it some more to get it just right…our advisor wants us to add some whiskers that illustratethe uncertainty. With the arrows() function we can take this plot to boomtown.

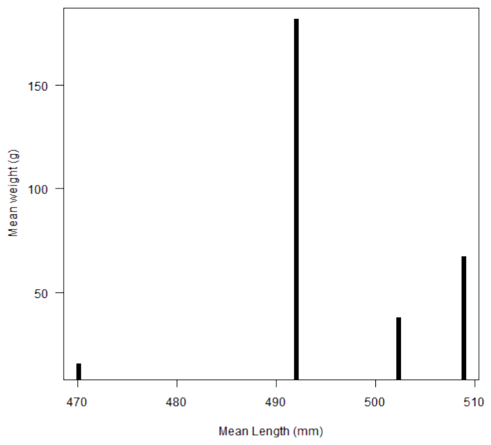
```
barplot(lindat$weight, names = lindat$year, las = 1, yaxt = "n",
ylim = c(0,250), ylab = "Mean weight (g)", width = 0.9, space =
0.1)
axis(side = 2, at = seq(0, 250, 50), labels = TRUE, las = 1)
box()
# dynamite plot
arrows(x0 = c(0.475, 1.475, 2.475, 3.475), y0 = lindat$weight,
x1 = c(0.475,
    1.475, 2.475, 3.475), y1 = lindat$uci, angle = 90)
```



Another example

Remember when we did some plots and changed the type from points, to lines to both? Well we can use 'h' to plot a histogram!

```
plot(weight ~ length, lindat, type = "h", lwd = 5, lend = 2, las
= 1, ylab = "Mean weight (g)", xlab = "Mean Length (mm)")
```



Argument breakdown:

type='h': plot histogram type bars

lwd=5: the width of those bars should be 5

lend=2: the cap of those bars should be square (1=rounded, 2=square)

Functions and repeated tasks

A lot of this stuff is pretty repetitive huh? Making use of functions can be very handy when there are multiple repetitive tasks.

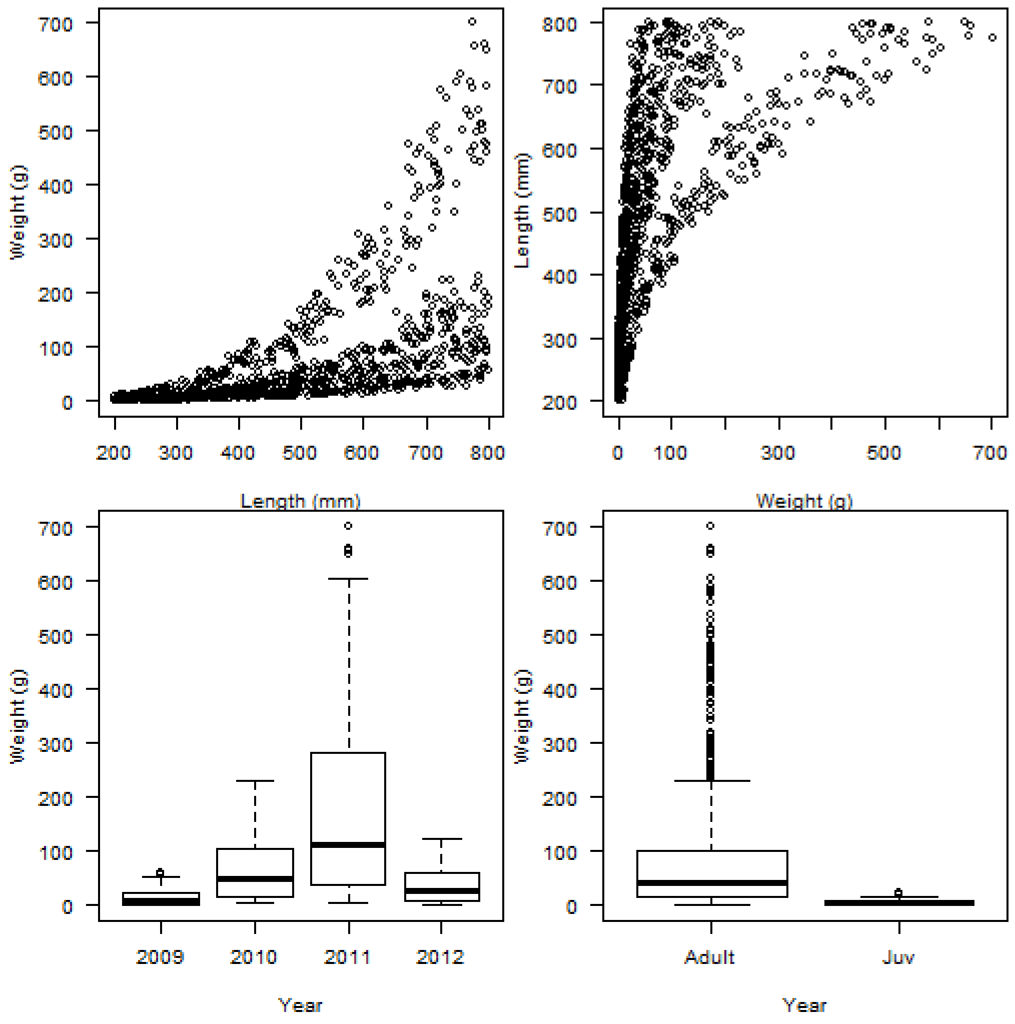
```
addpoints <- function(plotyear, plotsymbol, plotcol) {
    ## make a function to plot points for a given year
    points(weight ~ length, dat, subset = year == plotyear, pch
= plotsymbol, col = plotcol)
}
plot(weight ~ length, dat, type = "n", xlab = "Length (mm)",
ylab = "Weight (g)")
addpoints(plotyear = 2009, plotsymbol = 1, plotcol = "red")
addpoints(plotyear = 2010, plotsymbol = 2, plotcol = "blue")
addpoints(plotyear = 2011, plotsymbol = 2, plotcol = "black")
addpoints(plotyear = 2012, plotsymbol = 2, plotcol = "black")
```

Is there a way to convert the above to a “for loop”? Challenge accepted?

Using the **par()** function to create multi panel plots

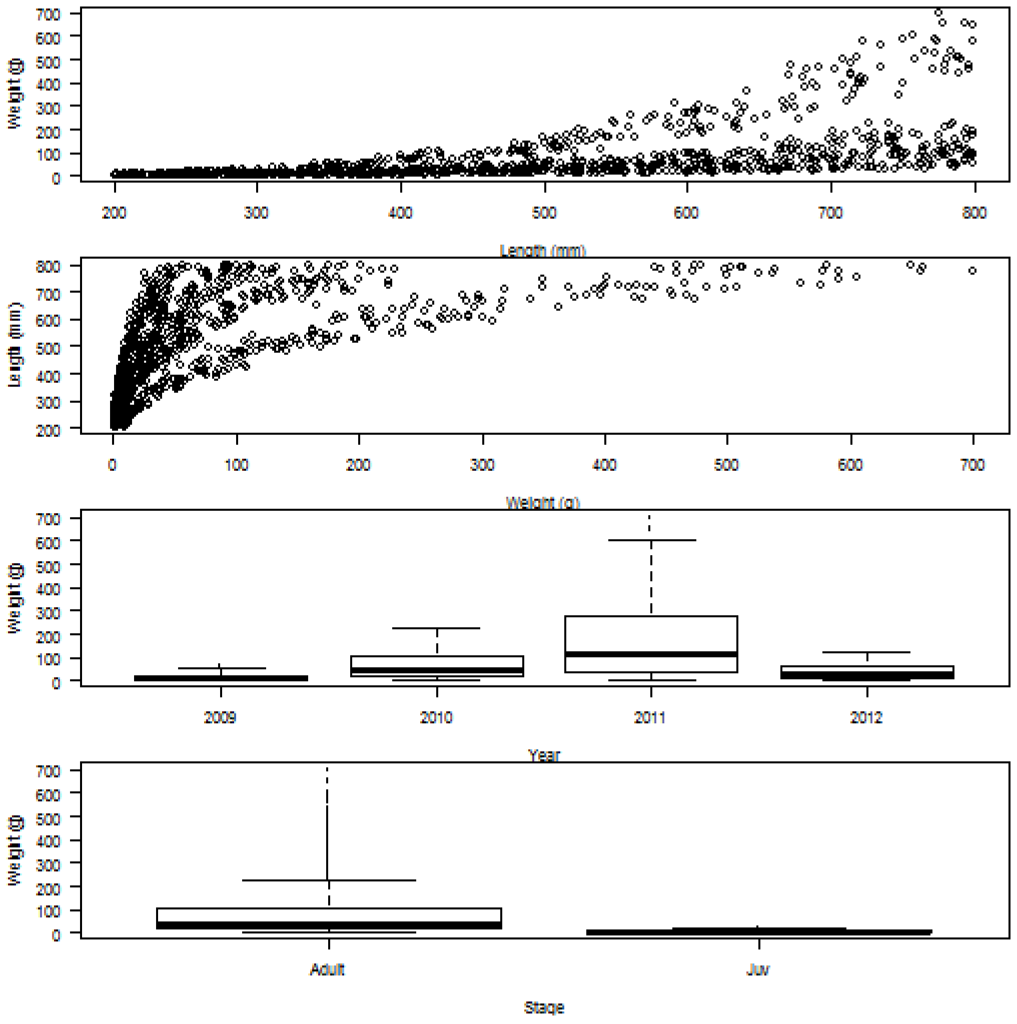
We can use the par() function to also set up the plotting lay out using the mfrow argument. It is pretty simple we can specify a plot that is 2 rows by 2 columns as

```
par(mfrow=c(2,2)). Lets give it a shot.
par(mfrow = c(2, 2), mar = c(4, 4, 0, 0.2))
plot(weight ~ length, dat, xlab = "Length (mm)", ylab = "Weight
(g)", las = 1)
plot(length ~ weight, dat, ylab = "Length (mm)", xlab = "Weight
(g)", las = 1)
boxplot(weight ~ year, dat, xlab = "Year", ylab = "Weight (g)",
las = 1)
boxplot(weight ~ stage, dat, xlab = "Year", ylab = "Weight (g)",
las = 1)
```


Weight (g)
Length (mm)
Length (mm)
Weight (g)
Weight (g)
Year
Weight (g)
Year
2009
2010
2011
2012
Adult
Juv

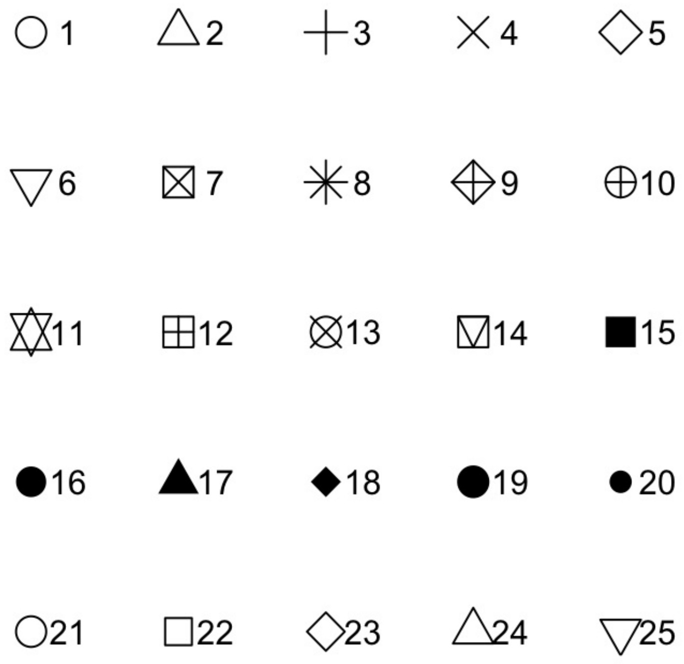
What about a 4 rows with 1 column? Yep we can do that! Just need to specify

```
par(mfrow=c(4,1))
par(mfrow = c(4, 1), mar = c(4, 4, 0, 0.2))
plot(weight ~ length, dat, xlab = "Length (mm)", ylab = "Weight
(g)", las = 1)
plot(length ~ weight, dat, ylab = "Length (mm)", xlab = "Weight
(g)", las = 1)
boxplot(weight ~ year, dat, xlab = "Year", ylab = "Weight (g)",
las = 1)
boxplot(weight ~ stage, dat, xlab = "Stage", ylab = "Weight
(g)", las = 1)
```



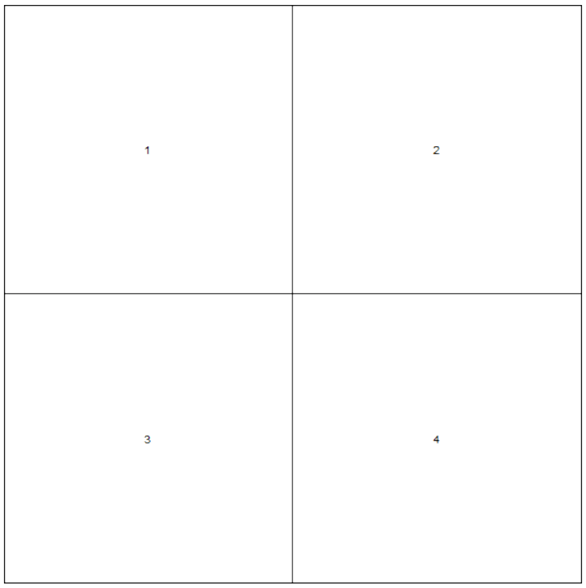
Additional material

Plot of pch plotting symbols



Using layout() to make multipanel plots

```
## LAYOUT() FUNCTION Figure 8.2 A: a 2x2 panel (2 rows and 2
columns)
layout(matrix(c(1, 2, 3, 4), 2, 2, byrow = TRUE))  # same as
par(mfrow = c(2, 2))
layout.show(4)  ## show the layout that has been set up for the
4 plots
```



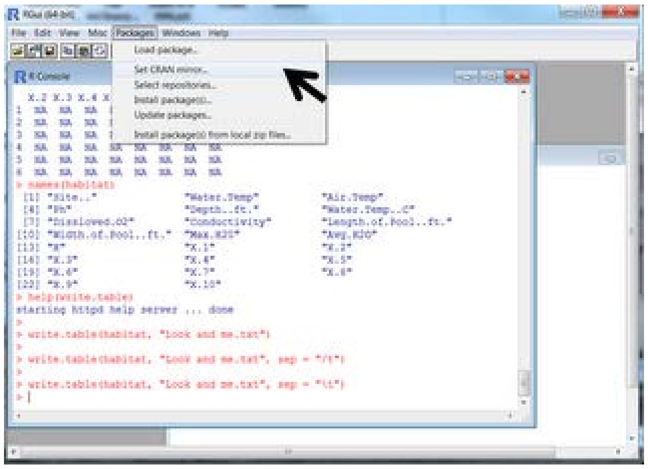
```
plot(weight ~ length, dat, xlab = "Length (mm)", ylab = "Weight
(g)", las = 1)
plot(length ~ weight, dat, ylab = "Length (mm)", xlab = "Weight
(g)", las = 1)
boxplot(weight ~ year, dat, xlab = "Year", ylab = "Weight (g)",
las = 1)
boxplot(weight ~ stage, dat, xlab = "Year", ylab = "Weight (g)",
las = 1)
## a 4x1 panel (4 rows and 1 column)
http://people.oregonstate.edu/~colvinmi/fw599/wk7/figure/unnamed
-chunk-412.png
layout(matrix(c(1, 2, 3, 4), 4, 1, byrow = TRUE))
layout.show(4)  ## show the layout that has been set up
par(mar = c(4, 4, 0, 0.2))
plot(weight ~ length, dat, xlab = "Length (mm)", ylab = "Weight
(g)", las = 1)
plot(length ~ weight, dat, ylab = "Length (mm)", xlab = "Weight
(g)", las = 1)
```

```
boxplot(weight ~ year, dat, xlab = "Year", ylab = "Weight (g)",
las = 1)
boxplot(weight ~ stage, dat, xlab = "Stage", ylab = "Weight
(g)", las = 1)
##  a different layout for 2 plots
layout(matrix(c(1, 1, 0, 2), 2, 2, byrow = TRUE))
layout.show(2)  ## show the layout that has been set up
plot(weight ~ length, dat, xlab = "Length (mm)", ylab = "Weight
(g)", las = 1)
boxplot(weight ~ year, dat, xlab = "Year", ylab = "Weight (g)",
las = 1)
##  Clean up he white space by adjusting the margins
layout(matrix(c(1, 1, 0, 2), 2, 2, byrow = TRUE))
layout.show(2)  ## show the layout that has been set up
par(mar = c(4, 4, 0.2, 0.2))
plot(weight ~ length, dat, xlab = "Length (mm)", ylab = "Weight
(g)", las = 1)
par(mar = c(4, 4, 0.5, 0.2))
boxplot(weight ~ year, dat, xlab = "Year", ylab = "Weight (g)",
las = 1)
```

INSTALLING AND LOADING R PACKAGES (example with internet connection)

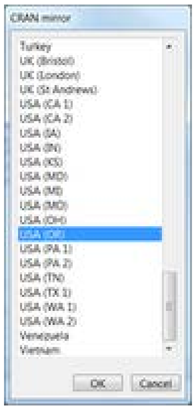
Basic R GUI

One of the beauties of R is the variety of packages for conducting statistical analyses, creating graphics, etc. These can be installed with either the basic GUI or R studio below. Unless you have the installation files (more below), you will need to have internet access. Let’s assume that you do have interet access and you want to install the “xlsx” package that allows you to import excel worksheets. We first need to install it on your computer and load it into the R environment. Why? Because it is an R package (a.k.a. R library). R packages are a set of functions that perform specialized tasks. The base R generally comes with several packages that are automatically loaded in the R environment every time you start it up. For example, the read and write commands are in the R base packages. In the base R GUI, we first to download the package from the internet from an R mirror site. These are various host institutions that have the packages available for downloads. OSU (go Beavs!) is one of the host institutions. To set the mirror with, base R GUI go to "Packages" and select "Set CRAN mirror...", e.g.,

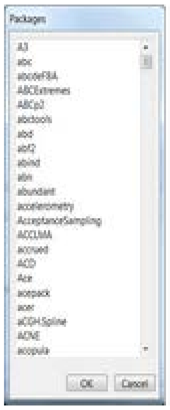


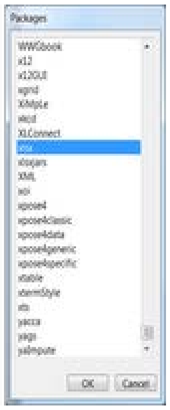
```
  X.2 X.3 X.4 X
1  NA  NA  NA
2  NA  NA  NA
3  NA  NA  NA
4  NA  NA  NA  NA  NA  NA  NA  NA
5  NA  NA  NA  NA  NA  NA  NA  NA
6  NA  NA  NA  NA  NA  NA  NA  NA
> names(habitat)
 [1] "Site.."              "Water.Temp"        "Air.Temp"
 [4] "Ph"                  "Depth..ft."        "Water.Temp..C"
 [7] "Dissloved.O2"        "Conductivity"      "Length.of.Pool..ft."
[10] "Width.of.Pool..ft."  "Max.H2O"           "Avg.H2O"
[13] "X"                   "X.1"               "X.2"
[16] "X.3"                 "X.4"               "X.5"
[19] "X.6"                 "X.7"               "X.8"
[22] "X.9"                 "X.10"
> help(write.table)
starting httpd help server ... done
>
> write.table(habitat, "Look and me.txt")
>
> write.table(habitat, "Look and me.txt", sep = "/t")
>
> write.table(habitat, "Look and me.txt", sep = "\t")
> |
```

Choose you're mirror site. We'll pick OSU as:



Click OK and the go to "Packages" and select "Install packag(e)" you will get the pop up to the left:





Scroll down and find xlsx and click OK. You will get the following (or something similar) in the console:

```
trying URL
'http://ftp.osuosl.org/pub/cran/bin/windows/contrib/3.0/xlsx_0.5
.1.zip'
Content type 'application/zip' length 314576 bytes (307 Kb)
opened URL
downloaded 307 Kb
package ‘xlsx’ successfully unpacked and MD5 sums checked
The downloaded binary packages are in
        C:\....some
filepath...\AppData\Local\Temp\Rtmp671J6Q\downloaded_packages
```

A zip file containing the package was downloaded to your computer in some default folder that will differ from the one shown above. Just make sure that you have the path right because you need to go there to install the package using the zip file. Go to "Package" and select "Install package(s) from local zip file". Go to the location indicated above

"`C:\ ...some filepath...\...downloaded_packages`" select the zip file and click "Open" and you will get the following:

```
package ‘xlsx’ successfully unpacked and MD5 sums checked
```

You are now ready to load the package. This can be done for any R package using one of two commands: library(package name) or require(package):

```
     # load xlsx package
     library(xlsx)
```

     or

```
     require(xlsx)
```

I generally prefer to use “require” because loading certain packages after they have already been loaded with library can produce screwy things. We usually load packages at the start of a session by including the library or require statements at the top of the R script. Note that these packages need to be reloaded an each time you begin an R session (but see setting R profile below for tricks).

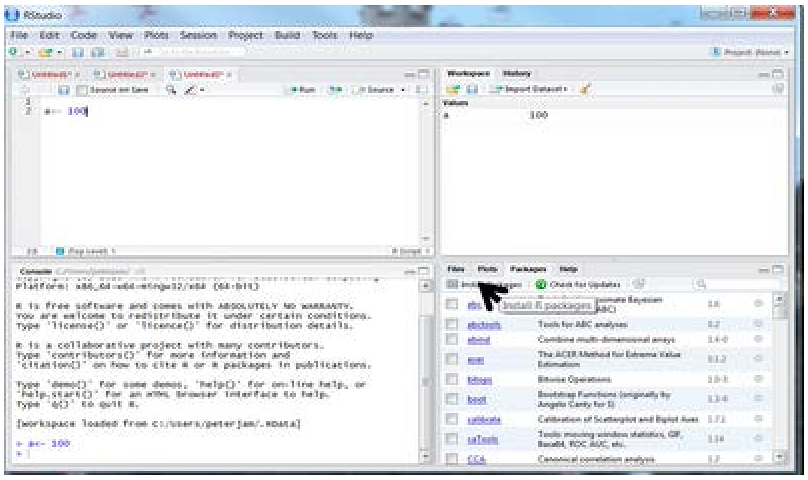
CEP loading packages: You may encounter the following (or similar) message when loading a package:

```
> require(jims.stuff)
Loading required package: jims.stuff
Warning message:
In library(package, lib.loc = lib.loc, character.only = TRUE,
logical.return = TRUE,  :
  there is no package called ‘jims.stuff’
```

SOLUTION: You don't have the package installed, so you need to install it using the above procedure.

Installation with Rstudio

Now let’s install xlsx package and read in the files using the R code above. The beauty of Rstudio is that it will automatically load packages for you. Just click the "Packages" tab and select "Install packages" as:



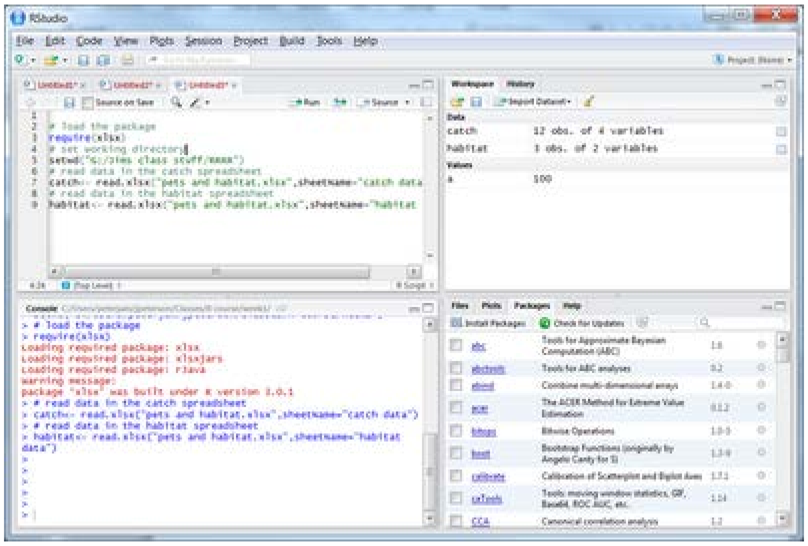
```
a<- 100
```

```
a                 100
```

```
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from C:/Users/peterjam/.RData]

> a<- 100
>
```

abc	[illegible] Bayesian [illegible] (ABC)	1.6
abctools	Tools for ABC analyses	0.2
abind	Combine multi-dimensional arrays	1.4-0
acer	The ACER Method for Extreme Value Estimation	0.1.2
bitops	Bitwise Operations	1.0-5
boot	Bootstrap Functions (originally by Angelo Canty for S)	1.3-6
calibrate	Calibration of Scatterplot and Biplot Axes	1.7.1
caTools	Tools: moving window statistics, GIF, Base64, ROC AUC, etc.	1.14
CCA	Canonical correlation analysis	1.2

A window will pop us and all you need to do is type the name of the package in the appropriate box and ta-da, it will install the package for you. You can load the package using the library or require commands as shown above of you can scroll down the packages list and check the box. It will then load the packages automatically. Go ahead and used the above code for reading in the Excel files. Be sure to include the code to load the xlsx package and set the working directory. Highlight the code in the script pane and click "Run" in the upper right hand corner of the script pane. The console should show the commands and the output from the commands just like it did in base R GUI. You should also should be able to see the names of the dataframes that you created in the Workspace pane, like this:



```
# load the package
require(xlsx)
# set working directory
setwd("G:/Jims class stuff/RRRR")
# read data in the catch spreadsheet
catch<- read.xlsx("pets and habitat.xlsx",sheetName="catch data
# read data in the habitat spreadsheet
habitat<- read.xlsx("pets and habitat.xlsx",sheetName="habitat
```

```
Data
catch        12 obs. of 4 variables
habitat      3 obs. of 2 variables
Values
a            100
```

```
> # load the package
> require(xlsx)
Loading required package: xlsx
Loading required package: xlsxjars
Loading required package: rJava
Warning message:
package 'xlsx' was built under R version 3.0.1
> # read data in the catch spreadsheet
> catch<- read.xlsx("pets and habitat.xlsx",sheetName="catch data")
> # read data in the habitat spreadsheet
> habitat<- read.xlsx("pets and habitat.xlsx",sheetName="habitat
data")
>
>
>
>
>
>
```

☐	abc	Tools for Approximate Bayesian Computation (ABC)	1.6
☐	abctools	Tools for ABC analyses	0.2
☐	abind	Combine multi-dimensional arrays	1.4-0
☐	acer	The ACER Method for Extreme Value Estimation	0.1.2
☐	bitops	Bitwise Operations	1.0-5
☐	boot	Bootstrap Functions (originally by Angelo Canty for S)	1.3-9
☐	calibrate	Calibration of Scatterplot and Biplot Axes	1.7.1
☐	caTools	Tools: moving window statistics, GIF, Base64, ROC AUC, etc.	1.14
☐	CCA	Canonical correlation analysis	1.2

Installation of an R package from a file

In this workshop, we will not have access to the internet but we will need to use a package (lmer) that does not come with the base software during the next lesson. Fortunately, we can install the package from a file. The files are located in the workshop materials in the file “lmer files”. You can install then in either the base R GUI or Rstudio by choosing install file and selection the option “install from package archive file” from the menu that pops up. Go to the location of “lmer files” and select all of the zipped files. You should be good to go.

BASIC STATISTICAL ANALYSIS WITH R

Linear modeling with r

Before this lesson, we need the following 2 comma separated text files:

Fish data.csv: contains lamprey collection data from 65 stream sections as part of a population survey. The file contains the following data:

Depth.m - depth of sample unit in meters

Velocity.ms - current velocity in m/s

Season - season sample was collected

Sample.area.m2 - sample unit area in meters squared

No.caught- the number of lamprey caught

Westslope.csv: contains presence and absence data from streams in watersheds within Interior Columbia River Basin. The file contains the following data:

PRESENCE- species presence (1) or absence(0)

WSHD - Watershed ID

SOIL_PROD - percent of watershed with productive soils

GRADIENT- gradient (%) of the stream

WIDTH- Mean width of the stream in meters

Don’t forget to set your working directory before you import the two files!

```
# set working directory
setwd("C:/Users/Jim/Desktop") #set working directory

# read in fish data and create data frame fish
fish<- read.csv("Fish data.csv")

# what’s in the data frame
head(fish)
```

```
# always a good idea to do a quick summary of the data before
analyses
summary(fish)
# always a good idea to calculate correlation between
potential independent
# variables in a regression here we do depth and velocity
column 1 and 2
cor(fish[,1:2])
```

We want to analyze fish density (lamprey per unit area) so we first transform the data and create a new variable.

```
## create density in fish data frame
fish$density.no.m2 <- with(fish,No.caught/Sample.area.m2)
```

We can conduct linear regression using several built in R functions. The first we will use is `lm()`. The syntax for `lm()` is: response variable, the curly sign (~) which means as a function of then the linear equation using the names of the independent variables, followed by the name of the data frame. For example, let's fit a model to see the relation between depth current velocity and lamprey density.

```
## model density as an additive function of depth and velocity
lm(density.no.m2~ Depth.m + Velocity.ms,data = fish)
```

We can create objects for all R statistical functions and conduct summaries on the objects. For example:

```
   #create mod.out to contain the results of the linear
   regression
   mod.out = lm(density.no.m2~ Depth.m + Velocity.ms,data = fish)

   #summarize the output, notice the increased information in the
   output
   summary(mod.out)
```

We also can enclose the model and the creation of the R object containing the output in a single step. Warning- this is one of the instances where = and <- are not treated the same.

```
## This will not work
summary(mod.out = lm(density.no.m2~ Depth.m + Velocity.ms,data =
fish))

## this works
summary(mod.out <- lm(density.no.m2~ Depth.m + Velocity.ms,data
= fish))
```

It is always good practice to query the objects created by R functions and see what is in them. The `lm()` function creates an `lm` object, which is basically a list.

```
# what is the object we created
class(mod.out)

## what us inside of the object
str(mod.out)
```

The contents and format of the `lm` object should look familiar. It is very similar to an R list and we can access the elements in the `lm` object just like we did in lists. For example,

```
# to extract the model coefficients (parameter estimates)
mod.out$coefficients

## we can extract the coefficients using shorthand
mod.out$coef
```

There also are built in functions that extract the objects that we want. For example,

```
# extract model coefficients
coef(mod.out)

## create 95% confidence intervals of the coefficients
(parameter estimates)
confint(mod.out, level = 0.95)
```

On occasion, it can be a pain to get standard errors from `lm` objects. One guaranteed way to do it is to extract the variance covariance matrix of any linear model (`lm`, `glm`, and other functions) using `vcov()`, grab the diagonal of the matrix and take the square root to get the standard errors.

```
# get standard errors for coefficients of model
sqrt(diag(vcov(mod.out)))
```

We can extract several other useful objects such as the residuals, the residual standard error, and predicted values.

```
# get residuals from mod.out
mod.out$residuals

# shorthand for residuals
mod.out$resid

# predicted (fitted) values of observations
mod.out$fitted.values

# shorthand for accessing predicted values
mod.out$fitted
```

We can use the data in the `lm` object to conduct some analyses. For example, we can conduct model diagnostics by examining residual plots.

```
# plot residual vs. predicted values
plot(mod.out$resid~mod.out$fitted)

# plot residuals vs. velocity notice two different data
sources
plot(mod.out$resid~fish$Velocity.ms)

# plot residuals vs. depth notice two different data sources
plot(mod.out$resid~fish$Depth.m)
```

The last plot should have caught your attention. It showed a curved relationship between the residuals and depth. This suggest that we may need to include a quadratic (squared) term in the regression. There are two basic ways of including quadratic terms in R statistical functions. The first is to create a new variable in the data frame which is the variable squared. For example,

```
fish$Depth.m.sq <- fish$Depth.m^2

#and use the new variable in the regression model
summary(mod.out <- lm(density.no.m2~ Depth.m+ Depth.m.sq +
Velocity.ms, data = fish))
```

A more efficient was is to use the `I()` function inside the `lm()` function. The `I()` function tells R that an operation needs to be performed inside the parenthesis before the model is fit.

```
summary(mod.out <- lm(density.no.m2~ Depth.m+ I(Depth.m^2) +
Velocity.ms, data = fish))
```

You can use `I()` in many other functions

Now, let’s examine our residuals to see if we’ve taken care of the problem.

```
plot(mod.out$resid~mod.out$fitted)

plot(mod.out$resid~fish$Depth.m)

# normal probability plot of residuals, should resemble a
straight line if model fit is OK
qqnorm(mod.out$resid)
```

Many times we may want to examine interactions between variables in statistical analyses. There are two ways to specify these interactions using an asterisk (*) and a colon (:)

```
# the asterisk indicates main effects and interactions between
2 or more variables
summary(mod.out <- lm(density.no.m2~ Depth.m*Velocity.ms, data
= fish))

# a colon means include the interaction between 2 or more
variables
summary(mod.out <- lm(density.no.m2~ Velocity.ms +
Depth.m:Velocity.ms, data = fish))
```

Notice above that the model included the interaction between depth and velocity but not the main depth effect. This is because we used the colon to specify the interaction. We're not sure how to interpret the parameter in this instance, so make sure you know what you are doing—statistically speaking.

In the summary of fish data frame, you should have noticed that season was a categorical predictor variable (a factor) that consisted of 3 levels spring, summer, and fall. Most R statistical functions can handle factor (categorical variables) by automatically creating binary indicator variables in the design matrix.

WHAT the #$@%^& did you just say?*

If we have time we will explain, suffice it to say that R chooses a baseline category (factor) by default usually in alphabetical order. Run the following and examine the output.

```
summary(mod.out <- lm(density.no.m2~ Depth.m+ I(Depth.m^2) +
Velocity.ms + Season, data = fish))
```

Fall was the first category alphabetically, so it was selected as the baseline in lm. The parameter estimates for spring and summer indicate how much lamprey density differed between spring and fall and summer and fall, respectively. There is a function for setting the order of the categorical variables—but we will not show it to you here. Rather, we suggest you create your own binary indicator variables using our friend the `ifelse()` function. Below we create a variable “fall”, which is = 1 if the season is fall and is zero of it is summer or spring.

```
# create binary indicator variable
fish$Fall = ifelse(fish$Season == "fall",1,0)
```

We can then use this variable in the model just like any other variable. Here the parameter estimate is an estimate of how much densities differed in the fall compared to spring and summer (model assumes spring and summer densities were not different).

```
summary(mod.out <- lm(density.no.m2~ Depth.m + I(Depth.m^2) +
Velocity.ms + Fall, data = fish))
```

There are several other functions for fitting linear models with R. `glm()` fits several types of generalized linear models, including regular (normal) linear regression that we did using lm. The syntax for `glm()` is similar to `lm()`. For example,

```
## now we try a different function glm that fits generalized
linear models
summary(new.out <- glm(density.no.m2~ Depth.m + I(Depth.m^2) +
Velocity.ms + Fall,data = fish))

## let’s see what's in the output list
str(new.out)
```

`glm()` function creates a glm object that is very similar to the object created with `lm()`. *Can you identify the differences?* There are several. One difference is that `glm()` provides us with AIC (Akaike Information Criteria) that can be used to determine the best predicting model. The data we have are relatively few, so we should be using AIC with a small sample bias adjustment AICc. To calculate AICc, we need AIC, the number of parameters in the model (stats heads call this the *model rank*) and the number of observations in the data set. If we look at the contents of new.out, we should be able to find most of these items. For example.

```
## first save AIC
aic = new.out$aic
## AICc requires number of observations in data we use
length() to get it
n = length(new.out$fitted)
## we also need the number of parameters in the model
## this is termed the rank we add 1 for the residual error
K = new.out$rank + 1
```

```
## here's how we calculate AICc
AIC.c <- aic + (2*K*(K+1))/(n-K-1)

## let’s put these in a vector
mod.sel = c(n,K,AIC.c)

#for grins lets fit another model without fall in it
new2.out <- glm(density.no.m2~ Depth.m + I(Depth.m^2) +
Velocity.ms,data = fish)

## lets calculate AICc for this model too notice that we
repeat the same commands as above
aic = new2.out$aic
n = length(new2.out$fitted)
K = new2.out$rank + 1

AIC.c2 <- aic + (2*K*(K+1))/(n-K-1)

## combine the AICc statistics for both models
rbind(mod.sel,c(n,K,AIC.c2))
```

That was pretty neat (ok…*we’re geeks*). By now, you should be able to see that it would be possible to create a function that extracts the elements necessary and calculates AICc.

The `glm()` function also fits other types of generalized linear models, i.e., linear models with other types of statistical distributions. Regular linear regression, i.e., the type of regression performed by `lm()` function, assumes a normal distribution which is often referenced as gaussian due to the use of the Gaussian function, see background here. Here’s how we would specify a normal distribution with `glm()`, unnecessary because gaussian is the default.

```
## specify distribution using family option in glm
summary(glm(density.no.m2~ Depth.m + I(Depth.m^2) +
Velocity.ms,data = fish, family = gaussian))
```

Poisson regression

This should produce results identical to above. If we look back at the fish data frame we see that the original response was No.caught, which are integers (whole numbers), e.g.,

```
head(fish)
```

We can model count (integer) data with GLMs using a Poisson statistical distribution. To do this we specify `family = poisson` in the `glm()` function.

```
## we specify this in glm using family = poisson
summary(pois.reg<-glm(No.caught ~ Depth.m + I(Depth.m^2) +
Velocity.ms + Sample.area.m2,data = fish, family = poisson))
```

We can examine the object created to hold the modeling information using the `str()` function.

```
## what's in the output, nothing new
str(pois.reg)
```

There is nothing different in the object from the normal linear regression with the exception of a `family = poisson` in the output. Just like normal linear regression, we can calculate AICc and examine the residuals using the $ syntax to access information in the object. For example, we can examine regression assumptions using a normal quantile-quantile (Q-Q) plot. If we meet model assumptions, the plot it should look similar to a straight line.

```
## normal quanntile plot of residuals
qqnorm(pois.reg$resid)
```

We can find out what other types of models `glm` fits using the `help()` function.

```
help(glm)
```

The help file indicates that we can fit several different types of GLMs. Of the remaining types (not used yet), logistic regression (`family = binomial`) is among the most widely used in ecological applications. To illustrate logistic regression, read in the westslope cutthroat trout presence/absence data.

```
# read in data create trout f=data frame
trout<-read.csv("Westslope.csv")

# examine contents of trout
head(trout)
```

Logistic regression

The response variable (dependent variable) in the data is PRESENCE (1 = present, 0 = absent) which is a binary response variable. Therefore, we need to use logistic regression and specify `family = binomial` in the `glm()` function.

```
## fit logistic regression model and create object logist.reg
summary(logist.reg<-glm(PRESENCE ~ SOIL_PROD + GRADIENT +
WIDTH,data = trout, family = binomial))

## what is in the object
str(logist.reg)
```

Similar to the objects created when conducting normal and Poisson linear regression, `logist.reg` contains familiar elements, such as parameter estimates, model rank, AIC, and residuals. We also access these using the $ syntax. For example, the westslope cutthroat trout data were collected from multiple streams within 56 watersheds. We have reason to believe that there may be dependence (autocorrelation) among streams within watersheds. To evaluate this, we could plot the residuals ordered by watershed number.

```
plot(logist.reg$resid~trout$WSHD)
```

The plot suggests that some watersheds have residuals that are much higher or lower than others. Let's create a box plot to make this much clearer.

```
boxplot(logist.reg$resid~trout$WSHD)
```

Hierarchical (mixed) models

These plots suggest strong spatial dependence among sites within watersheds. To account for the dependence, we could use hierarchical logistic regression. One useful package to fitting these models is `lme4`. Load the package.

```
require(lme4)
```

The `lme4` package fits generalized hierarchical linear models (there are others too) using the `glmer()` function. The syntax for the function is similar to `glm()` with a notable addition for specifying randomly varying effects. The code below specifies a model similar to the one fit above with `glm()`, but `(1|WSHD)` indicates that the intercept varies among watersheds. This is often called a random effect. Copy and paste the code and submit to R.

```
## fit logistic regression with random effect output to
H.logit
summary(H.logit <-glmer(PRESENCE ~ SOIL_PROD + GRADIENT +
WIDTH + (1|WSHD),data = trout, family = binomial))
```

This output differs from the `glm()` output. You should see new headings Random effects and Fixed effects and some familiar items, such as AIC. If time permits, we can discuss these outputs. Now it's time to see what is in `H.logit`. As above, we use the `str()` function to examine the contents.

```
str(H.logit)
```

Woa, that was different! *Did you notice the @?* This is because `lmer` creates a **S4 class** object. The above were **S3 objects**. There are lots of neat things that can be done with S4 objects, which is one of the reasons why many new packages create S4 objects, such as marked and unmarked. We won't be able to go into all of the features, so let's just figure out how to access the elements. Similar to the $ in S3 objects, we can use the @ to access S4 objects. We also can use familiar functions to access these elements. For example,

```
# use fitted function to grab predicted values
fitted(H.logit)
#use resid function to grab residuals
resid(H.logit)
```

We can also use the @ syntax within functions. Below we plot the residuals by watershed using a boxplot.

```
# boxplot of HLM residuals
boxplot(resid(H.logit)~trout$WSHD)
```

Did we take care of the spatial dependence? We can also create a Q-Q plot of the residuals.

```
#normal probability plot of residuals
qqnorm(resid(H.logit))
```

Similarly, we can extract the estimates of the fixed effects and their standard errors.

```
#extract fixed effect estimates
fixef(H.logit)
#extract fixed effect standard errors
sqrt(diag(vcov(H.logit)))
```

Hey, let's create a data frame with fixed effect parameter estimates, standard errors, and 90% confidence intervals.

```
## grab estimates and SE and combine
parms<-cbind(fixef(H.logit),sqrt(diag(vcov(H.logit))))

## Give the columns the correct names
colnames(parms)<- c("Estimate","Std.Error")

# coerce the matrix into a data frame
```

```
parms <- as.data.frame(parms)
# calculate lower and upper 90% CL using t-value 1.64
parms$Lower = parms$Estimate - 1.64*parms$Std.Error
parms$Upper = parms$Estimate + 1.64*parms$Std.Error
#print it out
parms
```

We can do additional analyses if time permits.

BASIC ECOLOGICAL SIMULATION

Ecological simulation can take many forms and we will not get even close to touching on them all. Simulation can be a powerful tool for understanding ecological dynamics and interactions, but it can also be an asset before you even go out and collect data!

Some quick review and useful tools

For loops redux

We learned about for loops earlier, remember that? Well for loops are a commonly used tool in simulation. You will see them in various simulation codes. But it doesn't stop there. You might also see a for loop in a for loop... What the what??? Basically, the outer for loop runs until it hits the inner for loop and repeats. Let’s work through this.

```
for(i in 1:3){
    cat("outer loop ", i, "\n")
    for (j in 1:5) {
        cat("inner loop ", j, "\n")
    }
}
```

Here is another example.

```
mns <- c()
```

```
for (i in 10:30) {
    sims <- c()
    for (j in 1:100) {
        x <- sum(rnorm(i, 80, 10))
        sims <- c(sims, x)
    }
    mns <- c(mns, mean(sims))
}
mns
```

Functions

We could also think about using a function to perform the same task.

```
sims <- function(i) {
    x <- c()
    for (j in 1:100) {
        x <- sum(rnorm(i, 80, 10))
        sims <- c(sims, x)
    }
    return(x)
}
```

We can feed this function into a for loop. Using a combination of for loops and functions can really help you keep your sanity if you are embarking on a complex simulation.

```
mns <- c()
for (i in 10:30) {
    mns <- c(mns, mean(sims(i)))
}
mns
```

Some basic simulation

Let's think about the model of lamprey density. Doing our literature review we have found there is a hypothesized association between depth and velocity. It would be prudent to do a bit of simulation to see how many sites we might need to visit to reliably estimate the strength of this association.

First we will simulate our covariates, depth and velocity.

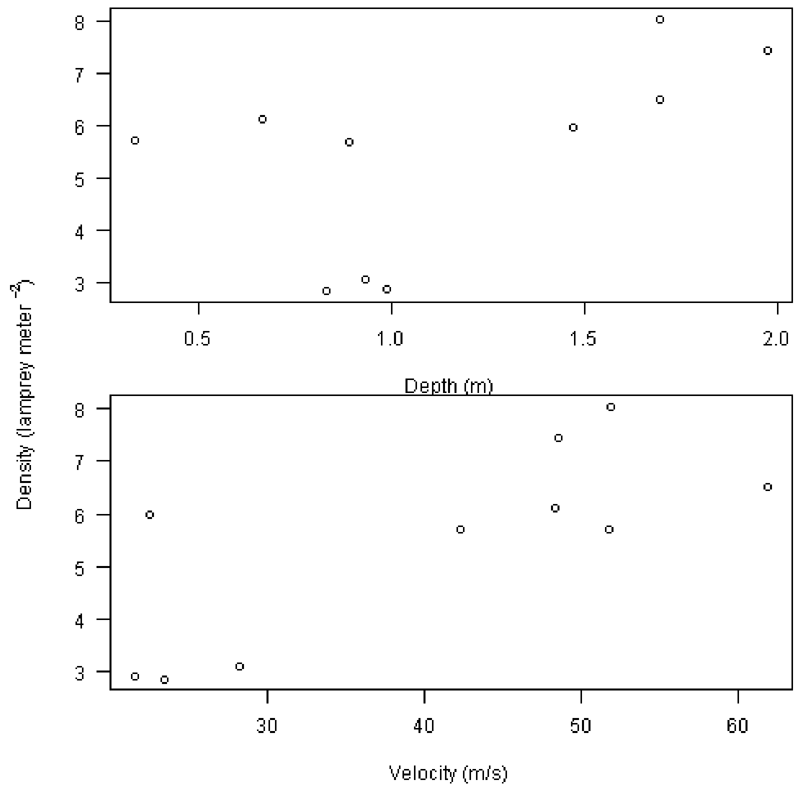
```
n <- 10
depth <- runif(n, 0.22, 2.2)
velocity <- runif(n, 0.03, 68)
B0 <- 0.237  # THESE VALUES FROM A BIT OF PILOT DATA
B1 <- -0.0063  # THESE VALUES FROM A BIT OF PILOT DATA
B2 <- 0.125856  # THESE VALUES FROM A BIT OF PILOT DATA
sim_density <- B0 + B1 * depth + B2 * velocity + rnorm(n, mean
= 0, sd = 1)
```

Another way to add error is as follows (since this approach is how you can simulate other distribution such as the Poisson or binomial).

```
pred <- B0 + B1 * depth + B2 * velocity + rnorm(n, mean = 0,
sd = 1)
sim_density <- rnorm(n, mean = pred, sd = 1)
```

Let’s look at what we created!

```
par(mfrow = c(2, 1), mar = c(4, 3, 0, 0), oma = c(1, 2, 1, 1))
plot(depth, sim_density, ylab = "", las = 1, xlab = "Depth
(m)")
plot(velocity, sim_density, ylab = "", las = 1, xlab =
"Velocity (m/s)")
mylab <- expression(paste("Density (lamprey meter "^-2, ")",
sep = ""))
mtext(side = 2, mylab, outer = TRUE)
```



```
sim_dat <- data.frame(depth = depth, velocity = velocity,
density = sim_density)
fit <- lm(density ~ depth + velocity, sim_dat)
summary(fit)
```

We can extract the standard errors and calculate the coefficient of variation as you learned above.

```
stderrs <- sqrt(diag(vcov(fit)))
ests <- coef(fit)
abs(stderrs/ests) * 100
```

Lets see what happens if we up n to 100

```
n <- 100
depth <- runif(n, 0.22, 2.2)
velocity <- runif(n, 0.03, 68)
sim_density <- B0 + B1 * depth + B2 * velocity + rnorm(n, mean
= 0, sd = 1)
sim_dat <- data.frame(depth = depth, velocity = velocity,
density = sim_density)
fit <- lm(density ~ depth + velocity, sim_dat)
summary(fit)
stderrs <- sqrt(diag(vcov(fit)))
ests <- coef(fit)
abs(stderrs/ests) * 100
```

Let’s see what happens if we up n to 200

```
n <- 200
depth <- runif(n, 0.22, 2.2)
velocity <- runif(n, 0.03, 68)
sim_density <- B0 + B1 * depth + B2 * velocity + rnorm(n, mean
= 0, sd = 1)

sim_dat <- data.frame(depth = depth, velocity = velocity,
density = sim_density)
fit <- lm(density ~ depth + velocity, sim_dat)
summary(fit)
stderrs <- sqrt(diag(vcov(fit)))
ests <- coef(fit) abs(stderrs/ests) * 100
```

We can take some of what we just did and roll it into a function that can be used in a bigger simulation.

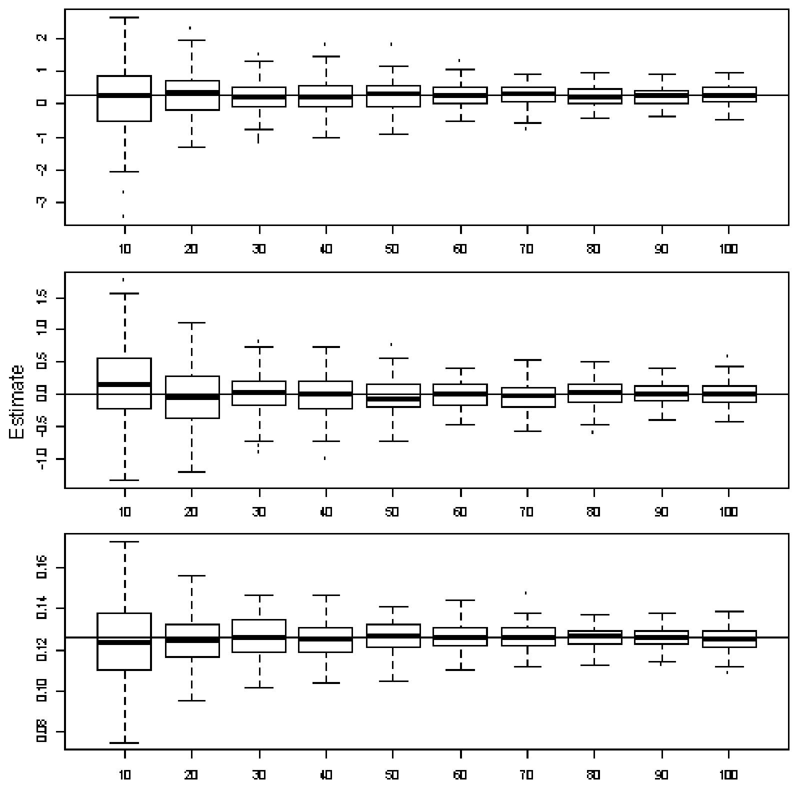
```
sim <- function(n = 20) {
    depth <- runif(n, 0.22, 2.2)
    velocity <- runif(n, 0.03, 68)
    B0 <- 0.237
    B1 <- -0.0063
    B2 <- 0.125856
    sim_density <- B0 + B1 * depth + B2 * velocity + rnorm(n,
mean = 0, sd = 1)
    fit <- lm(sim_density ~ depth + velocity)
    betas <- coef(fit)
    return(betas)
}
sim(n = 30)
```

Here we take that function and use a nested for loop to first set the sample size and then run 100 replicates for that sample size and collect the parameter estimates.

```
sampleSize <- seq(10, 100, by = 10)
output <- data.frame()
for (i in 1:length(sampleSize)) {
    for (j in 1:100) {
        betas <- sim(n = sampleSize[i])
        output_app <- data.frame(n = sampleSize[i], B0 =
betas[1], B1 = betas[2],
            B2 = betas[3])
        output <- rbind(output, output_app)
    }
}
head(output)
```

We can plot those results and see what happened in our simulation. You remember how to plot right?

```
par(mfrow = c(3, 1), mar = c(3, 1, 0, 1), oma = c(1, 3, 1, 1))
boxplot(B0 ~ n, output)
abline(h = 0.237)
boxplot(B1 ~ n, output)
abline(h = -0.0063)
boxplot(B2 ~ n, output)
abline(h = 0.125856)
mtext(side = 2, "Estimate", outer = TRUE, line = 1.5)
```



Poisson simulation

Here is an example for non-normal data. In this case we are simulating some counts assuming a Poisson distribution. The key here is how the Poisson distributed data is generated from the predictions using rpois()

```
#sample size
n <- 1000
#regression coefficients
beta0 <- 1
beta1 <- 0.2
beta2 <- -0.5
#generate covariate values
x <- runif(n=n, min=0, max=1.5)
x2<- runif(n=n, min=-3,max=3)
#compute mu's
mu <- exp(beta0 + beta1 * x + beta2*x2)
#generate Y-values
y <- rpois(n=n, lambda=mu)
#data set
data <- data.frame(y=y, x=x)
head(data)
glm(y~x + x2,data, family="poisson")
```

An exponential population model

The exponential model requires a single parameter, the intrinsic growth rate r. In addition, you need to give the population a place to start, the initial population size. Lets make this a bit more concrete with an example. An invasive species was recently introduced on Marys Peak. The Forest Service has estimated the abundance to be 2 plants per hectare. The intrinsic growth rate of 0.1 year^{-1} was estimated during a nearby invasion. Assuming that this invasion will be similar, the Forest Service would like to know how large the population will be after 10 years. Just to be clear the model parameters are as follows:

- $r = 0.1$ year^{-1}
- $N_0 = 2$ plants hectare^{-1}

Here is how we would go about doing this simulation using R. First we need to assign the objects that represent r, N_0, and the number of years to simulate as follows.

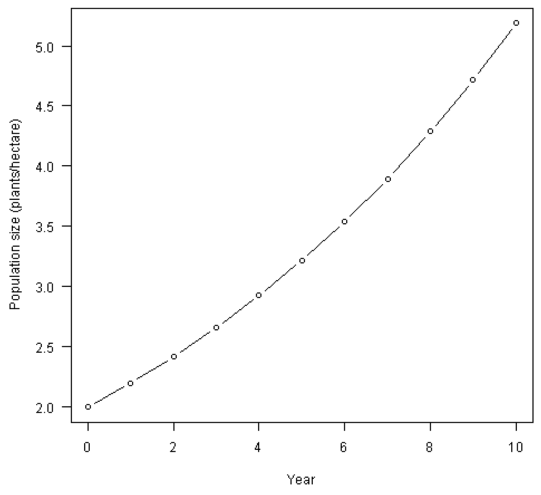
```
N0 <- 2
r <- 0.1
nyears <- 10
```

Next we are going to create an object to hold the simulated population densities. This is easily done by making a vector of NAs that is 11 elements long. Why 11? Well this is to allow us to set the initial population as the first element and then have 10 more elements to hold the simulated population.

```
N <- rep(NA, nyears + 1)
N <- N0
for (i in 1:nyears) {
    N[i + 1] <- r * N[i] + N[i]
}
N
```

And we can plot the simulation output…

```
plot(c(0:10), N, ylab = "Population size (plants/hectare)",
xlab = "Year", las = 1,
    type = "b")
```



There are several different ways to approach this problem in R, all producing the same results. Here are a couple ways you may see code specified if you happen to be Googling around. The approach below simulates from year 2 to 11 and simulates the population given the previous population size (i.e, N_{t-1}).

```
N <- rep(NA, nyears + 1)
N[1] <- N0
for (i in 2:(nyears + 1)) {
    N[i] <- r * N[i - 1] + N[i - 1]
}
N
```

The approach below uses the collection operator instead of creating a vector. One thing to consider is that this method can be very efficient in terms of computing time.

```
N <- N0
for (i in 1:nyears) {
    Nt1 <- r * N[i] + N[i]
    N <- c(N, Nt1)
}
N
```

Logistic population model

The exponential model is not a good approximation of population dynamics beyond the initial rapid colonization. The addition of a carrying capacity effectively allows the population to approach a maximum size. The equation is similar to the exponential model with the addition of an additional parameter K.

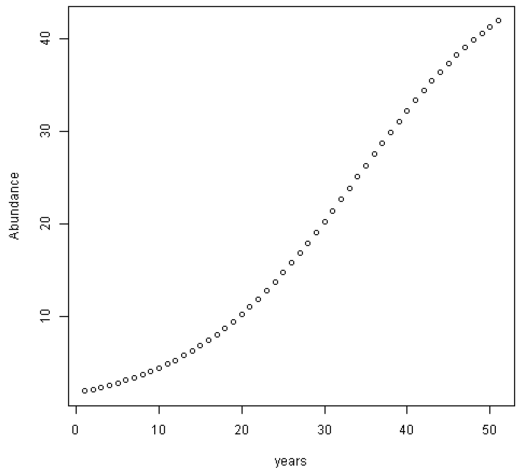
Building on the previous example, previous studies indicate that populations in similar areas never exceed 40 plants hectare$^{-1.}$ Just to be clear the model parameters are as follows:

- r = 0.1 year^{-1}
- N_0 = 2 plants hectare^{-1}
- K = 49

```
N0 <- 2
r <- 0.1
nyears <- 50
K <- 49
N <- rep(NA, nyears + 1)
N[1] <- N0
for (i in 1:nyears) {
    N[i + 1] <- r * N[i] * ((K - N[i])/K) + N[i]
}
N
```

Let's look at what we have going on here with this logistic model.

```
plot(c(1:51),N,ylab="Abundance",xlab="years")
```



Breaking the for loop

Sometimes in simulations you need to know when a condition occurs. Continuing with the invasion example, the Forest Service will apply an herbicide treatment when the population exceeds 30 plants per acre, but they need to know when this will happen given the model. A for loop can be stopped using the break argument.

```
N0 <- 2
r <- 0.1
nyears <- 50
K <- 49
N <- rep(NA, nyears + 1)
N[1] <- N0
for (i in 1:nyears) {
    N[i + 1] <- r * N[i] * ((K - N[i])/K) + N[i]
    if (N[i + 1] >= 30)
        break
}
N
```

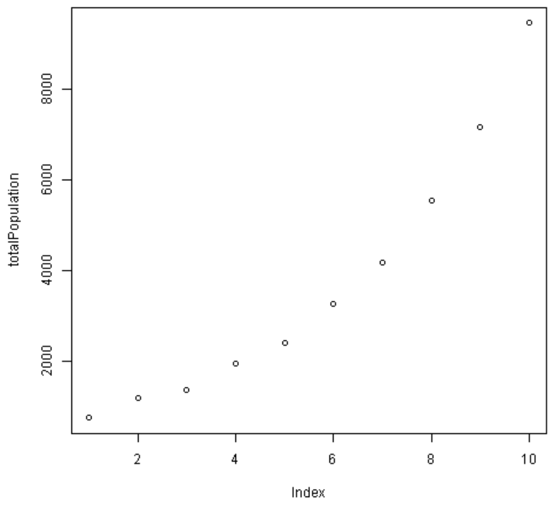
You can figure out how many years it took to exceed 30 plants hectare^{-1} using the length function omiting the NAs (of course you remember na.omit() right?).

```
length(na.omit(N))
```

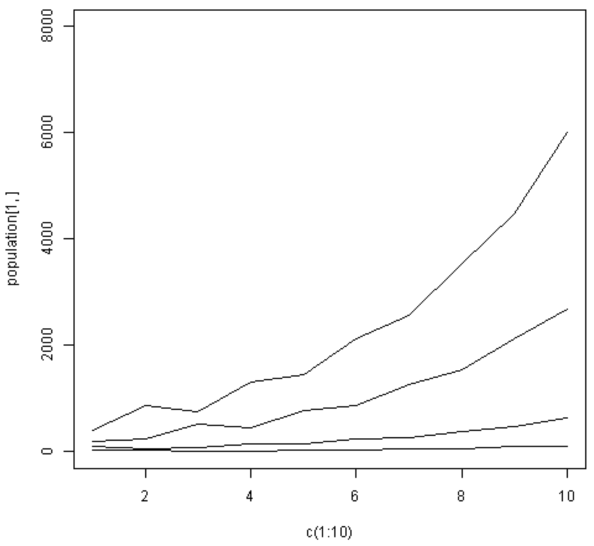
Age/stage structured population model

- Fecundity: Age 1 = 0, Age 2 = 2, Age 3 = 3, Age 4 = 4
- Survival: Age 1 to 2: 0.6, Age 2 to 3: 0.3, Age 3 to 4: 0.24

```
# AN AGE OR STAGE MATRIX OF FECUNDITIES AND SURVIVALS
A<- matrix(c(0,2,3,4, 0.6,0,0,0, 0,0.3,0,0,
0,0,0.24,0),nrow=4, byrow=T)

# A MATRIX TO HOLD THE FORECASTS OF THE POP
population<- matrix(0, nrow=4, ncol=10)

# SEED THE POPULATION
population[,1]<- c(400, 200, 100, 40)

# FORECAST THE STRUCTURED POPULATION
for(i in 2:10){
    population[,i]<- A%*%population[,i-1]
    }

# CALCULATE THE TOTAL POPULATION AT EACH TIME STEP
totalPopulation<- apply(population, 2, sum)
plot(totalPopulation)
```



```
plot(c(1:10),population[1,],ylim=c(0,8000),type='l')
points(c(1:10),population[2,],type='l')
points(c(1:10),population[3,],type='l')
points(c(1:10),population[4,],type='l')
```



```
# LAMBDA FOR MATRIX A
lambda<-as.numeric(eigen(A)$values[1])
w<-as.numeric(eigen(A)$vectors[,1])
w<-w/sum(w) # STABLE AGE DISTRIBUTION FOR MATRIX A
w
v<-as.numeric(eigen(t(A))$vectors[,1])
v<-v/v[1]; # REPRODUCTIVE VALUE FOR MATRIX A
v
s<-outer(v,w)/sum(v*w) # SENSITIVITY FOR MATRIX A
s
e<-(s*A)/lambda # ELASTICITIES FOR MATRIX A
e
```

ADDITIONAL R RESOURCES

Google Developers series of R tutorials:

https://www.youtube.com/playlist?list=PLOU2XLYxmsIK9qQfztXeybpHvru-TrqAP

If you have access to Lynda.com there are a couple of R courses

Coursera.org has a couple of online classes as well

100 Free R Tutorials: http://pairach.com/2012/02/26/r-tutorials-from-universities-around-the-world/