Applied Biostatistics - ECO 697

Introduction to R[†]

Abstract

The purpose of this lab exercise is to familiarize you with the `R` language and environment for statistical computing. Specifically, you will learn about variable types and data structures in `R`, vector and matrix operators in `R`, functions in `R`, getting data into and out of `R`, plotting in `R`, and libraries and packages in `R`.

† *This tutorial is only slightly modified from Kevin McGarigal's Introduction to* `R` *tutorial (found here).*

Contents

1 What is R?

R is a programming environment that uses a relatively simple language for data manipulation, calculation and graphical display. R is open source and is free to run, copy, and distribute. R comes with a number of basic tools that allow for data management, analysis, and plotting, but R also allows for user-created packages to be developed and distributed. In fact, much of the utility of R lies in these packages (of which there are over 6,000!), which are also open source. When a package is desired, it can simply be downloaded and installed in R (more about packages below).

2 Downloading and installing R

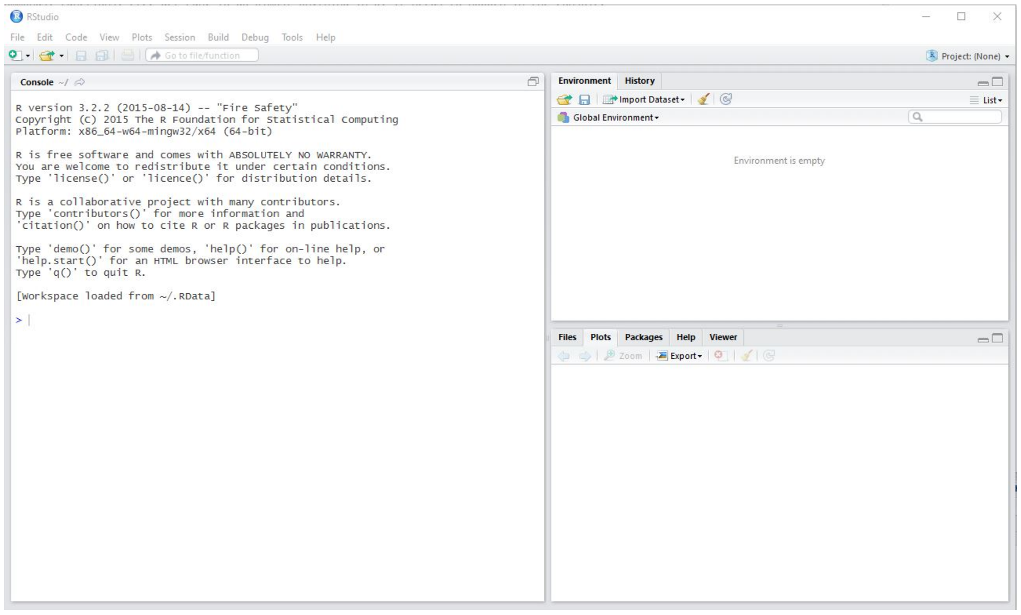
If you are running Windows, R can be downloaded here. Mac users can download R here (be sure you select the right version for your operating system). Install as you would any other software. Accepting the defaults is recommended.

3 Downloading and installing R Studio

You can open and work directly in the R software you just downloaded. However, there are many nicer user interfaces that make life with R a little easier. One of these is called R Studio. R Studio can be downloaded here (be sure you select the right version for your operating system). Install as you would any other software.

4 Opening an R session

Once you have R Studio installed, open it. You do not have to open R as R Studio will call it have it running in the background. When you open R Studio, you'll see something like this:



```
R version 3.2.2 (2015-08-14) -- "Fire Safety"
Copyright (C) 2015 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 ~/.RData]

> 
```

The console window on the left is where all the commands, functions, etc. get run. To do almost anything in `R`, it needs to happen in the console. On the upper right, you'll see two tabs, the *Environment* tab and the *History* tab. These are reduced in size and empty since nothing has been done in the `R` session yet. When we start running `R` script, the environment window will be populated with objects that we have created in our `R` session and are being saved in our session's memory. The history window will be populated with all the script we have run. It is worth noting here that everything in an `R` session is stored in virtual memory and this causes `R` to sometimes be memory limited – its only real drawback. If you run into memory issues, there are usually work-arounds, but this is for more advanced `R` usage. On the lower right, you'll see more tabs: *files*, *plots*, *packages*, *help*, and *viewer*.

Files: basically a mirror of your hard drive's directory.

Plots: where any plots you make will be displayed.

Packages: where the base packages included with the `R` software can be seen. This is also where any packages you install will be displayed. This window includes an easy 'Install' button that allows you to click it, type the name of the package you want to install and install it. It also includes a search window.

Help: where you can find help on any `R` packages and functions you have installed. The help tab includes a search window where you can type in any function name and its corresponding help file will appear.

Viewer: can be used to view local web content.

5 The R-script window and opening a .R file

R scripts are a handy way to work with and to save your R code. R scripts are created, viewed, and loaded in the script window ?? another window in R Studio we haven't seen yet. Any code in your script window can be run in the console. To open the script window for a new R script click on this icon at the upper left corner of R studio, then chose 'R script]:

Once your new script window is open, you'll see a blinking cursor. Type the following:

```
2+2
```

Then, with your cursor on the same line, click the 'Run' button in the upper right of the script window.

You'll notice that your code gets sent to the console and calculated:

```
2+2
```

```
[1] 4
```

You can run a single, or multiple, lines of code by highlighting it and clicking run. You can also run a line using keyboard shortcuts. For Windows users this is 'Ctrl + r' and for Mac it is 'Command +Return'. If you want to save your script, you can simply click the save button at the upper right of the script window, choose a destination, and name it. All R scripts are automatically given a .R extension. These can also be opened in any text editor and edited there. I wouldn't bother saving this one, but when you're ready to start writing your own script, this would be the way to start.

If you have a previously created script, like the ones we will be using in this lab for example, click the 'open file' button at the upper left of **R Studio**:

This will allow you to navigate to where you have saved the script. Navigate to the 'R.Intro.R' script and open it.

6 A few notes about the `R` language and the console

6.1 *The `R` language*

- `R` is case sensitive. If you name an object `MyModel`, then every time you want to refer to this object, you must type it with the two capital M's.
- Forward slashes, /, are used in all path names NOT backslashes
- Single and double quotes are used interchangeably as long as they're paired (e.g., "`stats`" or '`rules`')
- Square brackets, [ ], are used for indexing to specify a particular position in a vector (e.g., postition 6 of a vector called `vec`: `vec[6]`), or the row and/or column of a dataframe or matrix (e.g., row 6 column 5 of a dataframe called `df`: `df[6,5]`; the entire 15^{th} row: `df[15,]`; or the entire 2^{nd} column `df[,2]`)
- Parentheses, ( ), are used to contain the arguments of a function needed to successfully run that function
- Any text following a hashtag, #, is not read by `R` so in `R` scripts, the hashtag allows you to annotate the script (which is very mportant for taking notes in lab, or for when you return to a script at a later date. Notice the use of #'s in the *R.Intro.R* script.

6.2 *Console*

- You can run code in the console whenever you see the > sign.
- You'll notice above, that after we ran 2+2, `R` calculated the result and gave us the > sign again on the next line, indicating we can now run more code. Some things will take longer than this simple calculation. When they do, you won't see the > and a little stop sign will appear at the upper right of your console window. This means `R` is processing your calculations. When the processing in complete, you'll get the > sign back and you can run more code.
- Mistyping things in `R` is easy and sometimes code will be written incorrectly, for example, with a missing closing parenthesis. You will know there is something missing in the code when, on exectuting some code, you are left with a hanging plus sign, +, in the margin. This indicates that the code was incomplete in some way. For example type the following directly into the console and hit return:

  ```
  pi*(36/23
  ```

  See the + sign? Go ahead and add the missing parenthesisand hit return:

```
)
```

```
[1] 4.917275
```

- If you don't want to add anything after the + sign and want to get your > back press the Escape key.

7 R Objects and data types

Like most programming languages, `R` allows users to create variables or objects, which are essentially named computer memory. Objects are identified by a name assigned to them when they are created. Names should be unique, and long enough to clearly identify the contents of the `R` object (remember, these are case sensitive). You may work with the same data weeks or months later, so object names like `x` or `temp` are not very helpful. Object names can consist of letters, numbers, and the characters " . " and " _ ". They may not start with a number, nor can they include the character"$" or any arithmetic symbols as these have special meaning in `R` (e.g., +, −, * or).

Objects are assigned a value in an assignment statement, which in `R` has the variable name to the left of a left-pointing arrow, `<-`, with the value behind the arrow. For example, to set a variable that is the number of species in your dataset, you can do the following:

```
n.species <- 137
```

Go ahead and run this line from your script. From here on out, unless you reassign `number.species` to another value, `R` will equate `number.species` with the number 137.

`R` allows the creation of objects, which contain numeric values (both integers and floating point or real numbers), characters, or special characters interpreted as "logical" values. For example:

```
pi <- 3.14159
tiny.number <- 1.0e-10
species.name  <-'American robin'
conifer <- TRUE
```

Notice that real or floating point numbers can be entered with just a decimal point, or in exponential notation, where `1.0e-5` means 0.00001. Notice also that character variables, called *strings*, should always be typed inside of quotes (single or double, it doesn't matter as long as they match). Finally, note that the word `TRUE` is NOT surrounded by quotes. This is because in `R`, `TRUE` and `FALSE` (also `T` and `F`), are special values called logical expressions. Logical variables can only take the values TRUE or FALSE.

Unlike many programming languages (e.g., `FORTRAN` or `C`) you do not have to tell `R` what class an object is (integer, real, or character), and will thus only allo operations to be performed on particular variables. For example, here I will multiply a charachter by 37, which of course doesn't make sense, and `R` will tell us so:

```
species.name + 37
```

```
## Error in species.name + 37: non-numeric argument to binary operator
```

8 Data Structures

`R` is a 4^{th} generation language, meaning that it includes high-level routines for working with data structures, rather than requiring extensive programming by the analyst. In `R` there are 4 primary data structures we will use repeatedly.

- *vectors*: one-dimensional ordered sets composed of a single data type (i.e. a string of integers). Data types include integers, real numbers, and strings (character variables).
- *matrices*: two dimensional ordered sets composed of a single data type, equivalent to the concept of a matrix in linear algebra (i.e. a table of integers).
- *data frames*: single or multi-dimensional sets, and can be composed of different data types, although all data in a single column must be of the same type. In addition, each column and row in a data frame may be given a label or name to identify it. Data frames are equivalent to a flat file database, and similar to spreadsheets.
- *lists*: compound objects of associated data. Like data frames, they need not contain only a single data type, but can include strings (character variables), numeric variables, and even such things as matrices and data frames. In contrast to data frames, lists items do not have a row-column structure, and items need not be the same length; some can be a single values, and others a matrix. It's a little hard to imagine how lists operate in the abstract, but you will see that many of the results of analyses in `R` are returned as lists, so we'll introduce them as necessary.

8.1 Vectors and Matrices

Vectors, *matrices*, *data frames* and *lists* are identified by assigning a name to the data structure at the time it is created (like we did for `number.species` above). Names should be unique, and long enough to clearly identify the contents of the structure. Names can consist of letters, numbers, and the characters "." and " _ ". They may not start with a number, include the character"$", or include any arithmetic symbols as these have special meaning in `R`.

Vectors are often read in as data or produced as the result of analysis, but you can produce one simply using the concatenate function `c()`:

```
demo.vector1 <- c(1,4,2,6,12)
demo.vector.rand <- rnorm(10, 0, 1)
```

The above code created two vectors: `demo.vector1` which consisting of the numbers 1,4,2,6, and 12, and `demo.vector.rand` which is a vector of 10 numbers drawn at random from a normal distribution with mean 0 and a standard deviation of 1. Lets see:

```
demo.vector1
```

```
[1]  1  4  2  6 12
```

```
demo.vector.rand
```

```
 [1]  1.3383248  2.5401020 -1.0178287 -1.0503023  0.3213519  0.9449299
 [7] -0.7648863  0.9836316 -0.6135280 -0.5462825
```

As an aside, wrapping a statement in parentheses will print the object when you create it:

```
(demo.vector1 <- c(1,4,2,6,12))
```

```
[1]  1  4  2  6 12
```

```
(demo.vector.rand <- rnorm(10, 0, 1))
```

```
 [1] -2.6176985  1.3358021 -0.7517274 -0.3533199  1.5508308  2.0904533
 [7] -0.2265075 -1.5148133 -0.4892097  1.3955123
```

Individual items within a vector or matrix can be identified using indexing using square brackets, [ ]. For example, we can extract the n^{th} element of a vector where n can be any numeric integer between 1 and the number of elements in the vector ($n = 5$ for demo.vector1). Additionally, mutiple values can be extracted by specifying which elements using the concatenate function, `c()`, or as a continuous sequence like `[1:5]` or `[3:5]` etc.

Here's and example using the vectors we created earlier:

```r
#extract elements 1 THEN 4
demo.vector1[1]
```

```
[1] 1
```

```r
demo.vector1[4]
```

```
[1] 6
```

```r
#extract elements 1 AND 4
demo.vector1[c(1,4)]
```

```
[1] 1 6
```

```r
#extract elements 1 TO 4
demo.vector1[1:4]
```

```
[1] 1 4 2 6
```

A *factor* is a special type of vector object which is typically used to distinguish variables that can have a finite number of values (year, season, gender, age etc.). A factor object has a number of attributes that distinguish it from other object types. For example, a factor may be purely nominal or may have ordered categories. The importance of factors will become apparent when we start fitting models where they will influence the model fiting and parameter interpratation. However, it is important to recognize that any variable in a data set that contains a character (i.e., non-numeric value) will automatically be classified as a factor when imported into `R`.

A matrix can be created by simply binding together two or more vectors of the same type and length. For example, if we create a second vector, `demo.vector2` with values 4, 2, 1, 2, and 4, we can then bind the two vectors together using the `cbind()` (column bind) function to create a matrix with 5 rows and 2 columns (a 5×2 matrix).

```r
demo.vector2 <- c(4,2,1,2,4)
demo.matrix <- cbind(demo.vector1,demo.vector2)
```

In order to extract elemnet from, or subset a matrix, we still use square brackets, although now the row and column postition must be provided separated with a comma: `[rows,columns]`. Leaving either the first or second position blank selects all rows or columns, respectively.

```
#extract element in row 1 and column 2
demo.matrix[1,2]
```

```
demo.vector2
           4
```

```
#extract all elements in row 1
demo.matrix[1,]
```

```
demo.vector1 demo.vector2
           1            4
```

```
#extract all row elements in column 2
demo.matrix[,2]
```

```
[1] 4 2 1 2 4
```

```
#extract row elements 2 to 5 in column 1
demo.matrix[2:5,1]
```

```
[1]  4  2  6 12
```

Pasing empty values simply returns the origional matrix (or vector):

```
#matrices are all the same
demo.matrix
```

```
     demo.vector1 demo.vector2
[1,]            1            4
[2,]            4            2
[3,]            2            1
[4,]            6            2
[5,]           12            4
```

```
demo.matrix[]
```

```
     demo.vector1 demo.vector2
[1,]            1            4
[2,]            4            2
[3,]            2            1
[4,]            6            2
[5,]           12            4
```

```
demo.matrix[,]
```

```
     demo.vector1 demo.vector2
[1,]            1            4
[2,]            4            2
[3,]            2            1
[4,]            6            2
[5,]           12            4
```

```
#vectors are all the same
demo.vector.rand
```

```
 [1] -2.6176985  1.3358021 -0.7517274 -0.3533199  1.5508308  2.0904533
 [7] -0.2265075 -1.5148133 -0.4892097  1.3955123
```

```
demo.vector.rand[]
```

```
 [1] -2.6176985  1.3358021 -0.7517274 -0.3533199  1.5508308  2.0904533
 [7] -0.2265075 -1.5148133 -0.4892097  1.3955123
```

Rather than specifying which elements of a vector or matrix to extract, it is also possible to remove certain elements using the minus sign. `-` . For example, we could remove the 4th element of the `demo.vector1` vector like this:

```
demo.vector1[-4]
```

```
[1]  1  4  2 12
```

Remove the 3rd row of the matrix like this:

```
demo.matrix[-3,]
```

```
     demo.vector1 demo.vector2
[1,]            1            4
[2,]            4            2
[3,]            6            2
[4,]           12            4
```

Or, remove the 5th row and the 2nd column like this:

```
demo.matrix[-5,-2]
```

```
[1] 1 4 2 6
```

noting that, because the matrix only has two columns, this is equivalent to:

```
demo.vector1[-5]
```

```
[1] 1 4 2 6
```

(make sure you see why this is!).

8.2 Data Frames

Data frames can be manipulated in exactly the same way as matrices (see above), but can also be indexes using column/field/attribute names (I wil; stick with column names from here), which is useful as it does not require knowing the column number for a specific data item. To illustrate, let's load the 'birds.csv' dataset (note, we will come back to importing and exporting data later).

First, let's set the default working directory to the folder containing the 'birds.csv' file using the `setwd()` function. Note that the path specified should correspond to where the file is located on **your** machine. For example, windows users should have paths looking something like "C:/Users/yourname/.../R.intro", and Mac users should have paths looking something like "/Users/yourname/.../R.intro/".

```
setwd("C:/Users/chris/lab1/")
```

Next, read in the birds.csv dataset using the following code:

```
birds <- read.csv("birds.csv", header=TRUE)
```

Because the variable types are mixed in this incoming data set (containing both numeric and character fields), the data structure will be classed as a data frame automatically. Using the function `str()` we can inspect the data frame:

```
str(birds)
```

```
'data.frame':   32 obs. of  10 variables:
 $ BASIN: Factor w/ 2 levels "D","N": 1 1 1 1 1 1 1 2 2 2 ...
 $ SUB  : Factor w/ 2 levels "AL","BC": 1 1 1 1 1 1 1 1 1 1 ...
 $ BLOCK: int  1 2 3 4 5 6 7 8 9 10 ...
 $ AMGO : int  0 NA 0 0 0 1 0 0 0 0 ...
 $ AMRO : int  1 0 5 0 3 1 0 0 0 0 ...
 $ BCCH : int  0 0 2 0 0 0 0 0 0 0 ...
 $ BEKI : int  0 0 0 0 0 0 0 0 0 0 ...
 $ BEWR : int  0 0 0 0 1 0 0 0 0 0 ...
 $ BGWA : int  NA 0 2 2 2 2 2 2 2 2 ...
 $ BHGR : int  25 4 1 1 1 1 1 1 1 1 ...
```

For example, there is a column labeled 'AMRO' which is the abundance of **AM**erican **RO**bin for each sample plot (rows). We can extract this column in R as follows:

```
birds$AMRO
```

```
 [1] 1 0 5 0 3 1 0 0 0 0 1 2 0 2 1 1 0 0 1 0 0 0 2 0 0 0 0 0 0 0 0
```

where 'birds' is the name of the data frame and 'AMRO' is the name of the field or column of interest - the '$' is used to specify that we wish to extract a named field from the named dataframe.

8.3 Lists

As noted above, a list is a compound object composed of associated data. Items within a list are generally referred to as components. Similar to data frames, components in a list can be given a name, and the component can be specified by name at any time. In addition, components can be specified by their position in the list, similar to a subscript in a vector. However, in contrast to a vector or a matrix, the position of components in a list are specified using double square brackets, [[ ]]. We will ultimately find it quite handy to create our own lists, but for the first few labs we will just see them as results from analyses, so we'll take them as they come and demonstrate their properties by example.

Here is a very simple example using the objects we created above, and for illustration, the american robin abundance values.

```
list.demo <- list(demo.vector1, demo.vector2,
                  demo.matrix, birds$AMO)

names(list.demo)<-c('vec1','vec2', 'mat1', 'AMRO.abund')

list.demo
```

```
$vec1
[1]  1  4  2  6 12
```

```
$vec2
[1] 4 2 1 2 4

$mat1
     demo.vector1 demo.vector2
[1,]            1            4
[2,]            4            2
[3,]            2            1
[4,]            6            2
[5,]           12            4

$AMRO.abund
NULL
```

The point here is that lists can be used to store multiple objects of various classes and dimensions as a single data object.

9 Checking objects

All data objects (vectors, matrices, data frames and lists) have an internal structure that defines the type of object (e.g., data frame) and its components. It is important to always be aware of the structure of the objects you are working with, and this is especially so with data frames which can contain mixtures of data types (e.g., integers, real numbers, factors etc) and lists which can contain mixtures of object types.

One way to view the structure of an object is, as we saw above, is the `str()` function. For example, inspecting the structure of the birds data frame:

```
str(birds)
```

```
'data.frame':   32 obs. of  10 variables:
 $ BASIN: Factor w/ 2 levels "D","N": 1 1 1 1 1 1 1 2 2 2 ...
 $ SUB  : Factor w/ 2 levels "AL","BC": 1 1 1 1 1 1 1 1 1 1 ...
 $ BLOCK: int  1 2 3 4 5 6 7 8 9 10 ...
 $ AMGO : int  0 NA 0 0 0 1 0 0 0 0 ...
 $ AMRO : int  1 0 5 0 3 1 0 0 0 0 ...
 $ BCCH : int  0 0 2 0 0 0 0 0 0 0 ...
 $ BEKI : int  0 0 0 0 0 0 0 0 0 0 ...
 $ BEWR : int  0 0 0 0 1 0 0 0 0 0 ...
 $ BGWA : int  NA 0 2 2 2 2 2 2 2 2 ...
 $ BHGR : int  25 4 1 1 1 1 1 1 1 1 ...
```

the resulting output tells us that the object is a data frame containing 32 observations (rows) and 10 variables (columns). Furthermore, it lists each variable, the associated class or data type (e.g., factor or integer in this case), and the first 10 values. For factors, it also gives the number of levels (i.e., unique values) and what they are.

For large data frames, it is sometimes useful to view just a portion of the object to inspect it. Two functions, `head()` and `tail()`, will print out the first six and last six records of the object, respectively, although you change the number of records to print by adding an argument to the function. For example, we can print out the first 8 records of the birds data frame as follows:

```
head(birds,8)
```

```
  BASIN SUB BLOCK AMGO AMRO BCCH BEKI BEWR BGWA BHGR
1     D  AL     1    0    1    0    0    0   NA   25
2     D  AL     2   NA    0    0    0    0    0    4
3     D  AL     3    0    5    2    0    0    2    1
4     D  AL     4    0    0    0    0    0    2    1
5     D  AL     5    0    3    0    0    1    2    1
6     D  AL     6    1    1    0    0    0    2    1
7     D  AL     7    0    0    0    0    0    2    1
8     N  AL     8    0    0    0    0    0    2    1
```

Lastly, it is sometimes useful to view the entire dataset in a spreadsheet-like format. The function `fix()` can be used for this purpose. This function opens the data object in a data editor window that has the look and feel of a spreadsheet. You can browse the object and/or edit the values. Importantly, when you close the editor, any changes that you made will be preserved in the object's in memory. However, changing values in the object stored in memory does not change the original data file stored on disk, if there is one. The change is made to the object stored in memory and is preserved only for the duration of the `R` session. Try it out:

```
fix(birds)
```

10 R Operators

Because R is a 4th generation language, it is often possible to perform fairly sophisticated routines with very little programming. The key is to recognize that R operates best on vectors, matrices, or data fames, and to capitalize on that. A large number of functions exist for manipulating vectors, and by extension, matrices and data frames (usually). For example, birds is a bird abundance data frame containing 32 sample plots (rows) and 10 fields (columns). The first 3 fields (columns) contain plot identifiers. The first two are character fields (BASIN and SUB) and the third field is numeric (BLOCK). The remaining 7 fields (columns) are numeric and contain abundances for 7 different bird species. Given this data structure, we can perform the following:

```
max(birds[,5]) # maximum value of 2nd species (5th column) among all plots.
```

```
[1] 5
```

Alternatively, because birds is a data frame, we can accomplish the same thing with the following:

```
max(birds$AMRO) #same!
```

```
[1] 5
```

We could compute the total number of American robins observed across all plots:

```
sum(birds[,5])
```

```
[1] 20
```

```
sum(birds$AMRO) #same
```

```
[1] 20
```

or apply a log transofmration to American robin (AMRO) and Black-throated gray warbler (BGWA) counts from plots 4 through 10. Notice two things here, first, I have used the `round(x,2)` function to round the whole output to 2 decimal places, and second i have domonstrated how character strings can also be used for indexing data frames.

```
round(log(birds[4:10,c(4,9)]+0.1),2) # +0.1 to avoid NaN
```

```
   AMGO BGWA
4  -2.3 0.74
5  -2.3 0.74
```

```
6   0.1 0.74
7  -2.3 0.74
8  -2.3 0.74
9  -2.3 0.74
10 -2.3 0.74
```

```r
round(log(birds[4:10,c("AMRO","BGWA")]+0.1),2) # same!
```

```
    AMRO BGWA
4  -2.30 0.74
5   1.13 0.74
6   0.10 0.74
7  -2.30 0.74
8  -2.30 0.74
9  -2.30 0.74
10 -2.30 0.74
```

In addition, `R` supports logical operators when indexing, where the the positions returned are those for which the logical statement equaks `TRUE`. Logical operators include:

‘R‘	Meaning
`>`	*greater than*
`>=`	*greater than or equal to*
`<`	*less than*
`<=`	*less than or equal to*
`==`	*equal to*
`!=`	*not equal to*
`&`	*and*
`|`	*or*

Let’s look at some of these in practice. We could ask how many non-zero counts American robin counts there were across all the sites (i.e., plots with counts greater than 0) by doing the following:

```r
sum(birds[,5]>0)
```

```
[1] 11
```

However to understand what is happening here, perhaps it’s useful to break this down further, and mention that `R` treats `TRUE` values as 1 and `FALSE` values as 0:

```
(greaterthan0 <- birds[,5]>0)
```

```
 [1]  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE
[12]  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
[23]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
sum(greaterthan0)
```

```
[1] 11
```

Here are a couple more examples. In how many plots were exactly 5 black-headed grosbeak (BHGR) observed?

```
sum(birds[,8]==5)
```

```
[1] 0
```

```
sum(birds$BEWR==5)
```

```
[1] 0
```

Or, what was the maximum number of black-headed grosbeak (BHGR) in sites where at least one of the following three species were observed: American goldfinch (AMGO), American robin (AMRO), or black-capped Chickadee (BCCH). This is a bit more tricky, but note the use of the *or* operator):

```
max(birds[,"BHGR"][birds$AMGO>0|birds$AMRO>0|birds$BCCH>0])
```

```
[1] NA
```

We get an NA. Thats common - when doing a calculation, an `NA` in the data will typically results in an `NA` being returned (more on this next). However, most functions have an `na.rm` arguement which instructs the runtion to ignore `NA`'s when doing the calculation:

```
max(birds[,"BHGR"][ birds$AMGO>0|birds$AMRO>0|birds$BCCH>0],na.rm=T)
```

```
[1] 25
```

10.1 Missing Values

Missing values in `R` are defined using an `NA`. Missing values, or `NA`'s, are common in ecological data sets and can cause problems if not dealt with correctly. We'll discuss how to deal missing values in more detail in the next lab, but for now lets assume that we want to simply drop the missing values from a vector. To do this we could define a new vector, say `x`, which has the `NA` removed. We can use the `is.na()` function to identify which element is an `NA` and the *not* operator to select all elements that not *not* `NA`'s. The fourth column from the birds data frame contains a single missing value so lets try that:

```
(x<-birds[,4][!is.na(birds[,4])]) #or
```

```
 [1] 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
(x<-birds[!is.na(birds[,4]),4])   #same!
```

```
 [1] 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Again this can be broken down if you didnt follow what's happening here:

```
is.na(birds[,4])
```

```
 [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
!is.na(birds[,4])
```

```
 [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[23]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

It is important to note that applying logical operators to identify `NA`'s doesnt always work (e.g., x[x!='NA'] will not work), so it's best to use the specialized `NA` handling functions and arguements.

Since dropping missing records is a very common operation, `R` has a built in function, `na.omit()`, that makes it even easier to accomplish the same thing as above:

```
(na.omit(birds[,4]))
```

```
 [1] 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
attr(,"na.action")
[1] 2
attr(,"class")
[1] "omit"
```

```
(birds[,4][!is.na(birds[,4])]) #see!
```

```
 [1] 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

This use of missing values is critical to `R` because all operations on vectors or matrices must have the same number of elements. So, if there are missing values in any field we're using in a calculation, the same record (row) must be omitted from all the other fields as well.

11 Subseting Matrices or Data Frames

It is frequently useful, or even necessary, to subset a data set by selecting certain columns, rows or both. For example, you may need to select a set of numeric variables (columns) from a data frame containing a mixture of variable types in order to conduct subsequent analyses that require numeric data. Or you may wish to select a set of observations (rows) that meet certain criteria. There are many ways to subset a data set depending on whether you are selecting columns or rows and whether the data structure is a matrix or data frame. Here we will assume that the initial data structure is a data frame, since this is what we will typically be working with. We could use the indexing like we did before:

```
# all rows and columns 4 through 10
birds.new <- birds[,4:10]
birds.new <- birds[,-c(1:3)] # same

# plots where SUB is AL and all columns
birds.new<-birds[1:16,]
birds.new<-birds[-c(17:32),] # same
```

Or, we could use the `subset()` function to get the same result:

```
# all rows and columns 4 through 10
birds.new <- subset(birds,select=AMGO:BHGR) #same as above
birds.new <- subset(birds,select=c(AMGO,AMRO,BHGR)) # same

# plots where SUB is AL and all columns
birds.new <- subset(birds,subset=SUB=='AL')
```

To subset by rows *and* columns, you can just combine any of the above.

12 Row or Column Operations on a Matrix or Data Frame

Vector operators can be applied to every row or column of a matrix to produce a vector with the `apply()` function (see `?apply()`). This function works by first supplying the matrix, specifying the direction, or margin, of the function call (across row: 1, or down columns: 2), and then the function you wish to apply across rows or columns. Lets try using this function to find the maximun count of each species across the 32 plots:

```
(bird.max <- apply(birds[,4:10],2,max))
```

```
AMGO AMRO BCCH BEKI BEWR BGWA BHGR
  NA    5    2    0    1   NA   25
```

Ah, but there's those pesky `NA`'s again! In the `apply()` function, any arguments that you would usually pass to the function (here `max`) can be passed after the function as additional agguments; `R` knows they are calls to the internal function. So, lets fix that:

```
(bird.max<-apply(birds[,4:10],2,max,na.rm=TRUE)) # 2 for col-wise apply
```

```
AMGO AMRO BCCH BEKI BEWR BGWA BHGR
   1    5    2    0    1    2   25
```

Now let's see what the maximum count was at each plot (remember plots are the rows):

```
(bird.sum<-apply(birds[,4:10],1,sum,na.rm=TRUE)) # 1 for row-wise apply
```

```
 [1] 26  4 10  3  7  5  3  3  3  3  5  5  6  8  6  6  5  5  6  7  7  3  9
[24]  7  7  7  1  3  3  7  5  5
```

We can also use an expression within the `apply()` function. For example, to sum the number of plots where species counts are greater than 0 we could do this:

```
(bird.sum<-apply(birds[,4:10]>0,2,sum,na.rm=TRUE))
```

```
AMGO AMRO BCCH BEKI BEWR BGWA BHGR
   1   11    3    0    2   20   32
```

Here, because of the command `birds[,4:10]>0`, the `apply()` function is being applied to a matrix of TRUE (1) and FALSE (0) values created from columns 4 through 10 of the birds data frame.

13 Functions in `R`

A function consists of a name and one or more parameters (or arguments) contained in parentheses that are required to process the function. A simple function that we have already used is the `sum()` function, which returns the sum of all the values present in its arguments. In its simplest, `sum()` contains two arguments: `sum(x, na.rm=FALSE)`, where the first argument, `x`, is the data (either a vector, matrix, or data frame containing all numeric variables), and the second argument indicates whether missing values should be ignored or not. The default `na.rm=FALSE` will return `NA` if there are any missing values, whereas `na.rm=TRUE` will ignore the missing values when calculating the sum (we saw this above). In addition, functions will have default values for additional options/arguents so there is no need to include the arguments if you wish to use the defaults.

If we apply the `sum()` function to the entire data frame of counts, the sum is across all elements in all rows and in columns 4 through 10:

```r
sum(birds[,4:10],na.rm=TRUE)
```

```
[1] 190
```

The `apply()` function we used above is a special function that allows us to apply other functions to each column or row of the matrix. In this case, we applied the `sum()` function to each species in the birds data set and returned a vector of values containing the sum of abundance for each species.

When using functions it is important to understand the arguments of the function. The arguments of a function are all defined in the associated help file (see below). Each function has one or more named arguments. Some or all of the arguments may come with default values, in which case you do not need to specify any arguments inside the () when calling the function. However, in most cases one or more of the arguments will not have a default value and thus you must provide a value for the argument. For example, the `sum()` function requires that you specify a data set (an object, either a vector, matrix, or data frame containing all numeric variables). If you do not specify a value for this argument, you will get an error message.

In addition, if you specify values for arguments in the order that they are given in the written function, then the arguments do not need to be named explicitly in the function call. For example, in the `apply()` function, the following two calls are equivalent:

```
apply(X=birds[,4:10], MARGIN=2, FUN=sum)
```

```
AMGO AMRO BCCH BEKI BEWR BGWA BHGR
  NA   20    4    0    2   NA  123
```

```
apply(birds[,4:10],2,sum)
```

```
AMGO AMRO BCCH BEKI BEWR BGWA BHGR
  NA   20    4    0    2   NA  123
```

This is because in the second call the arguments are given in the same order as expected. If, however, you want to specify the arguments in a different order from the default, then the argument names must be included in the function call, e.g.:

```
apply(birds[,4:10], FUN=sum, MARGIN=2)
```

```
AMGO AMRO BCCH BEKI BEWR BGWA BHGR
  NA   20    4    0    2   NA  123
```

In practice, explicitly naming the arguments often is required when you want to use a mixture of default and optional arguemnts. In general, I find it good practice to always name the arguments in a function call to make it explicit what you are doing, albeit at the cost of verbosity in your code. Functions are essential to working with R. You will be using functions constantly to manipulate, summarize, analyze, and graphically display your data. For most of the things you will need to do in this course, functions have already been written by others and you will simply need to know how to call these functions and interpret their output. However, you can't work long in R without confronting the need to construct your own functions. In most cases, these will be functions that call or make use of existing R functions, but in particular ways suited to your applications.

Throughout this course, we will make extensive use of existing R functions to complete projects, and there may be a need or opportunity for you to create your own functions. Any time you issue a set of commands that you anticipate having to repeat or reuse in the future, you should consider writing a function. Although we will not go into the details of writing functions here, you can easily review the code for a function by simply typing the function name at the console. Finally, if in doubt about functions, always consult the help pages, one way of doing this is using a question mark followed by the function name (see more below):

```
?apply()
```

14 Getting Data Into and Out of R

Getting data into any program is often the hardest part about using the program. For R, this is generally not true, as long as the data are reasonably formatted. The R *Development Core Team* has developed a special manual to cover the ins and outs of getting data into and out of R. It's available as a PDF or HTML here.

Typically a data set would be organized in columns, with column headings, and blanks or tabs between entries. Like this for example:

BASIN	SUB	BLOCK	AMGO	AMRO	BCCH	BEKI	BEWR	BGWA	BHGR
D	AL	1	0	1	0	0	0	NA	25
D	AL	2	NA	0	0	0	0	NA	4
D	AL	3	0	5	2	0	2	NA	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
N	BC	14	0	0	0	0	0	2	5
N	BC	15	0	0	0	0	0	0	5
N	BC	16	0	0	0	0	0	0	5

The columns do not need to be straight, but multi-word variables like *clay loam* need to be connected or put in quotes. For example, multiple words that should be treated as a single charactercan be connected using periods (*clay.loam*) or underscores (*clay_loam*). Words cannot be connected with the dollar sign as the $ symbol has a special meaning (see above). The above file (called *birds.dat* which you should have) could be read into R using the `read.table()` function. Note that the default for for the `header=` option in `read.table()` is FALSE which states that the columns *don't* have names and that the first row of the data frame is data. This is rarely the case in my experience, and if columns *do* have names/headers, we must set `header=TRUE`. We can do all of this like so:

```
birds <- read.table('.../birds.dat', header=TRUE)
```

where '.../' is replaced with the full path to the folder containing the *birds.dat* file ion your computer. Alternatively, it is customary to set the working directory at the beginning of a session using the `setwd()` function. This saves from having to write out the entire path when reading tand writing files. An **example** might be:

```r
setwd('c:/work/stats/ecodata/lab/introduction to R/')
```

In the above code, the data frame in `R` resulting from reading in the *birds.dat* file would be named `birds`, and the columns would be named exactly as in the data file. Note that the value for BGWA in the third plot is NA. `R` automatically treats these values as missing data, but if you use other codes for missing values, you specify them in the `read.table()` function . For example, suppose in your data set you used -999 as the missing value code; adding an additional arguement, `na.strings`, tells `R` to treat any cell with an entry -999 as missing data and it will will be converted to `NA`. We would do that as follows:

```r
bogus <- read.table('bogus.dat',header=TRUE,na.strings='-999')
```

In fact, data can be stored in a number of formats, all of which can be imported into `R` using a `read.`*something*`()` function. The most common formats are text files (.txt) or comma separated files (.csv), excel files are a little more challenging so I tend to avoid reading in data form those. The .txt files are read in exactly as above. For .csv files we can use the `read.table()` function and specify the seaprator using the argument `sep=',',` or the `read.csv()` function. Conveniently, the default `header=` option in `read.csv()` is TRUE, so we dont need to write that out (but we will here, for completeness, we actually did this earlier):

```r
birds <- read.table('birds.csv',header=TRUE,sep=',')
birds <- read.csv('birds.csv') # same
```

The row names and column names can be looked at or changed using the `row.names()` and `colnames()` functions respectively:

```r
#look at the names
row.names(birds)
```

```
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14"
[15] "15" "16" "17" "18" "19" "20" "21" "22" "23" "24" "25" "26" "27" "28"
[29] "29" "30" "31" "32"
```

```r
colnames(birds)
```

```
 [1] "BASIN" "SUB"   "BLOCK" "AMGO"  "AMRO"  "BCCH"  "BEKI"  "BEWR"
 [9] "BGWA"  "BHGR"
```

```r
#change the names
row.names(birds) <- row.names(birds) #set to what they are already
```

```
colnames(birds) <- colnames(birds) #set to what they are already
```

The great benefit of the `read.table()` function is the way it handles variables. If any value in a column is alphabetic, it treats the column as a *factors* or *categorical* variable. There is never a need to convert a categorical variable to numeric. However, if you already have categorical variables coded as integers, you can explain that to `R` with the `as.factor()` function after you read the data in. If all values in a column are numeric, it treats that variable as numeric. For example, let's say we wanted to convert the BLOCK field from an integer (numeric) to a factor (categorical), since it represents plot ID and does not contain any quantitative information:

```
birds$BLOCK <- as.factor(birds$BLOCK)
```

To verify that we made the desired change, let's check the structure of the birds object using the `str()` function, as follows:

```
str(birds)
```

```
'data.frame':   32 obs. of  10 variables:
 $ BASIN: Factor w/ 2 levels "D","N": 1 1 1 1 1 1 1 2 2 2 ...
 $ SUB  : Factor w/ 2 levels "AL","BC": 1 1 1 1 1 1 1 1 1 1 ...
 $ BLOCK: Factor w/ 16 levels "1","2","3","4",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ AMGO : int  0 NA 0 0 0 1 0 0 0 0 ...
 $ AMRO : int  1 0 5 0 3 1 0 0 0 0 ...
 $ BCCH : int  0 0 2 0 0 0 0 0 0 0 ...
 $ BEKI : int  0 0 0 0 0 0 0 0 0 0 ...
 $ BEWR : int  0 0 0 0 1 0 0 0 0 0 ...
 $ BGWA : int  NA 0 2 2 2 2 2 2 2 2 ...
 $ BHGR : int  25 4 1 1 1 1 1 1 1 1 ...
```

Note that BLOCK is now a factor with 16 levels. Alternativewly, we an ask for the class of BLOCK:

```
class(birds$BLOCK)
```

```
[1] "factor"
```

Getting data out of `R` is just as easy as getting data into `R`, if not easier. While there are numerous options for outputting `R` objects, here we will focus on the most common task, outputting a data frame using the generic `write.table()` function. For simplicity, let's take the birds data set that we just read into `R` and stored in the object named *birds* and write it back out as a comma-delimited file:

```r
write.table(birds, 'myNewFile.csv', row.names=FALSE, sep=',')
```

This statement will write the object birds to a file named *myNewFile.csv* in the current working directory (have a look). Note, if you haven't set the working directory to the desired location using the `setwd()` function, you can include the full path in the `write.table()` function.

A few things about the `write.table()` statement above function are worth noting. First, by default, `write.table()` will overwrite a file by the same name if it exists in the output directory (so be careful in naming the output file!). However, if you wish to append the object to the existing file, you can add the `append=TRUE` argument in the statement which will append the new file (data frame) to the botton of the existing one. Second, in the statement above, I set the `row.names=FALSE` (the default is TRUE) which will omit the row names from the output file. Third, I also included a `sep=',' ` argument to specify explicitly that I want to use commas as column delimiters in the output file. As with everything in `R`, there are several ways to do anything. For writing comma-delimited output files, you can also use the `write.csv()` which always uses a comma delimiter. There are lots of other options for outputting data frame using the `write.table()` function, so, as always, it is worth familiarizing yourself with the help file for this function: `?write.table`.

Finally, sometimes it is useful to write out `R` objects as *.RData* objects to save writing and reading files repeatedly. For example, you can save the `R` version of ther data for use at a lter time as follows:

```r
save(birds, file= "c:/work/stats/ecodata/lab/R.intro/birds.RData")
```

Then we can simply open it again from within `R`:

```r
load("c:/work/stats/ecodata/lab/R.intro/birds.RData")
```

Note: No matter what you named the file to disc, when `R` reads it back in it will contain the same name as was in `R` memory. For example, if you wrote out birds as birds_out.RData, when you read it back in, it will still have it's original name: *birds*.

15 Plotting in R

R has a powerful graphics capability that is much of the appeal to using the system. Many of the analyses have special plotting capabilities that allow you to plot results without storing multiple intermediate products. R supports a fairly broad range of graphic devices in addition to excellent on-screen plotting. Reflecting its origins on unix computers, it is quite good at Postscript output, but also includes other formats. The devices available to you for plotting will depend to some extent on your operating system (Windows versus unix/linux). R includes postscript, pdf, pictex, and xfig as vector devices, and png and jpeg as raster (pixel) devices. Simply type:

```
?Devices #or
help(Devices)
```

to get a list of available devices and their names. Each of the devices has options that can be set to control plot size, orientation (landscape or portrait), font size, etc. To get a quick feel for how easy it is to create plots, let's first create a simple data set containing three numeric variables:

```
x <- sort(runif(25,-2,2))
y <- rnorm(25, 20 + 4.5*x, 2) #does this look familiar?
```

Now we can produce a simple scatter plot of x against y using the basic `plot()` function:

```
plot(x,y) # you'll see the plot in the plot tab in \texttt{R} Studio
```

We can change just about any aspect of the plot with a bewildering array of graphical controls given as arguments to the plot function. Here are some examples

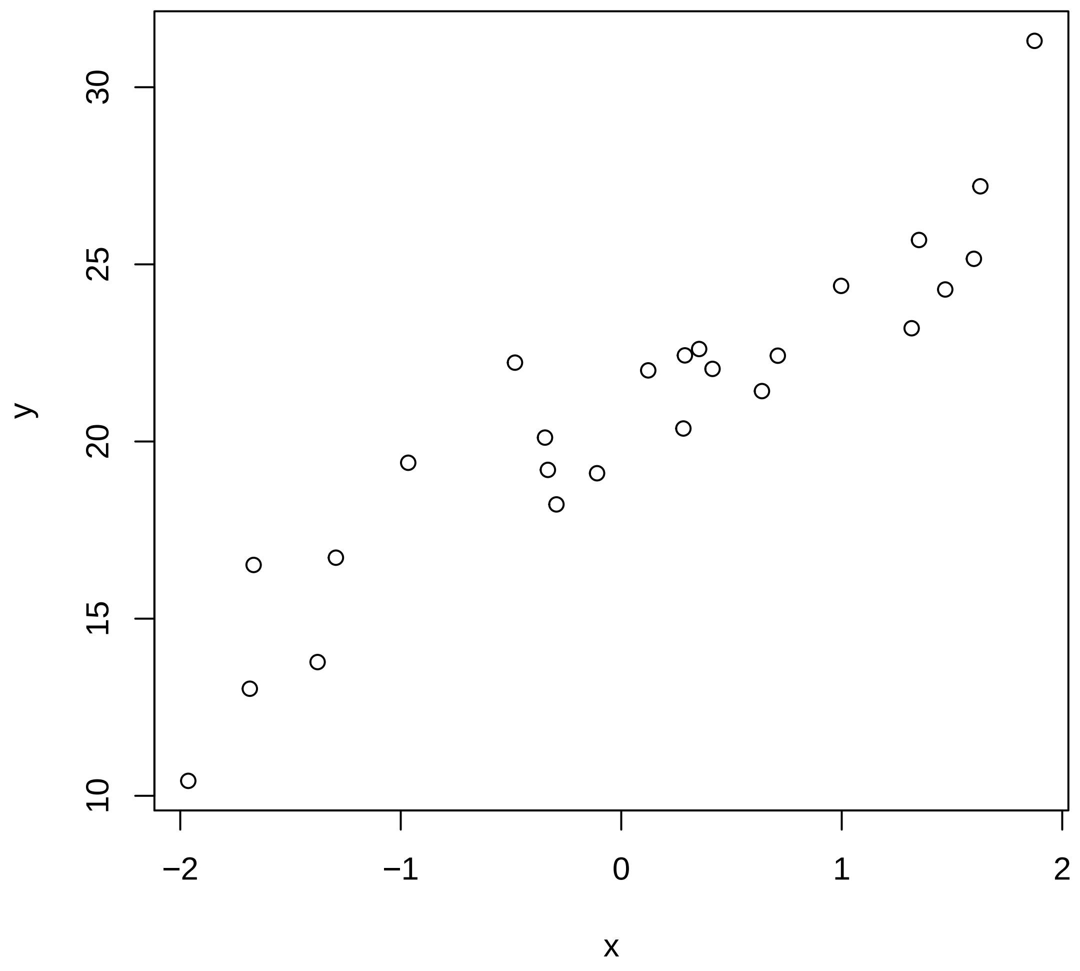


Figure 1: Default plotting style

We can change what *type* of plot we want to use:

```
#explore different plot 'type'
par(mfrow=c(2,2))
plot(x,y,type='l')
plot(x,y,type='o')
plot(x,y,type='b')
plot(x,y,type='h')
```

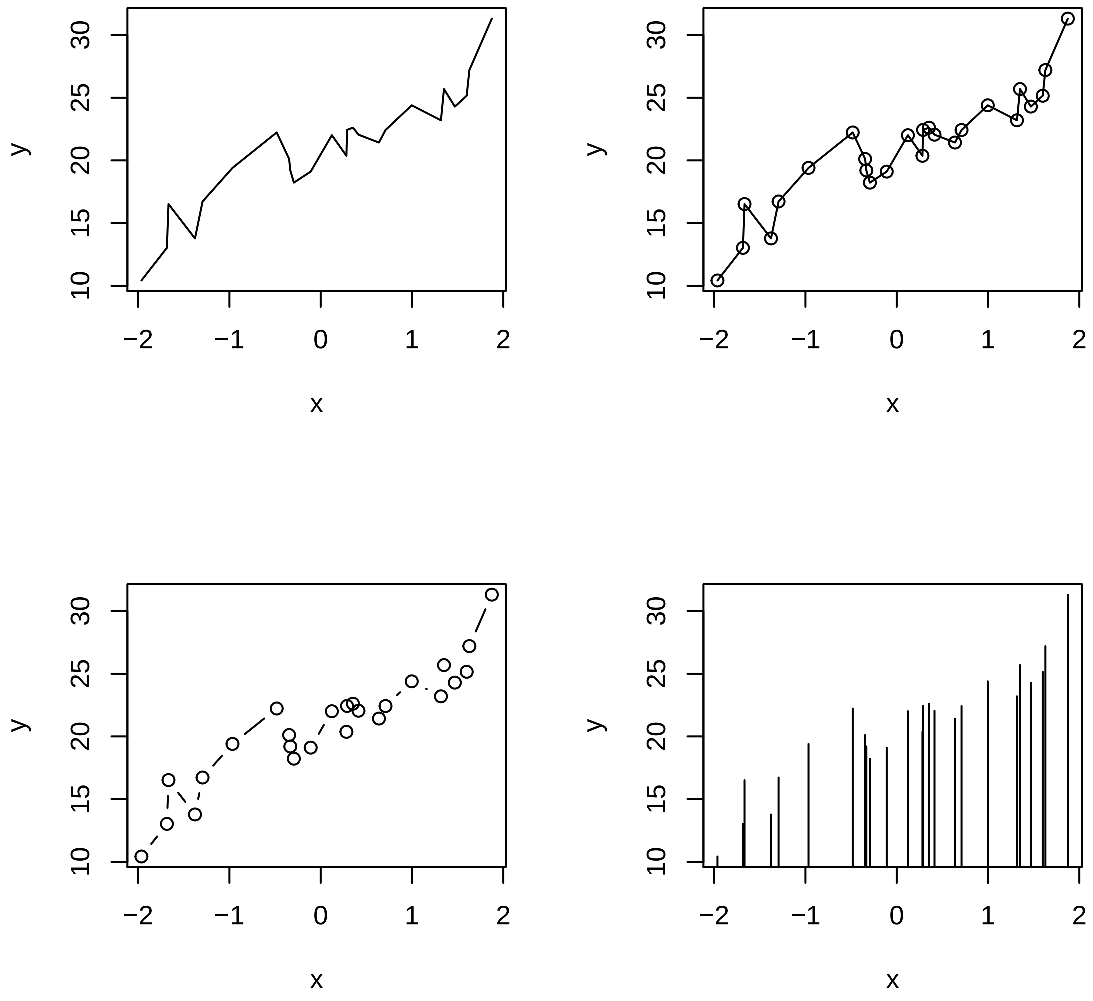


Figure 2: Alternative plotting types

We can change what *line type* (`lty`) we want to use:

```r
#explore different plot 'type'
par(mfrow=c(2,2))
plot(x,y,type='l', lty=1)
plot(x,y,type='l', lty=2)
plot(x,y,type='l', lty=3)
plot(x,y,type='l', lty=4)
```

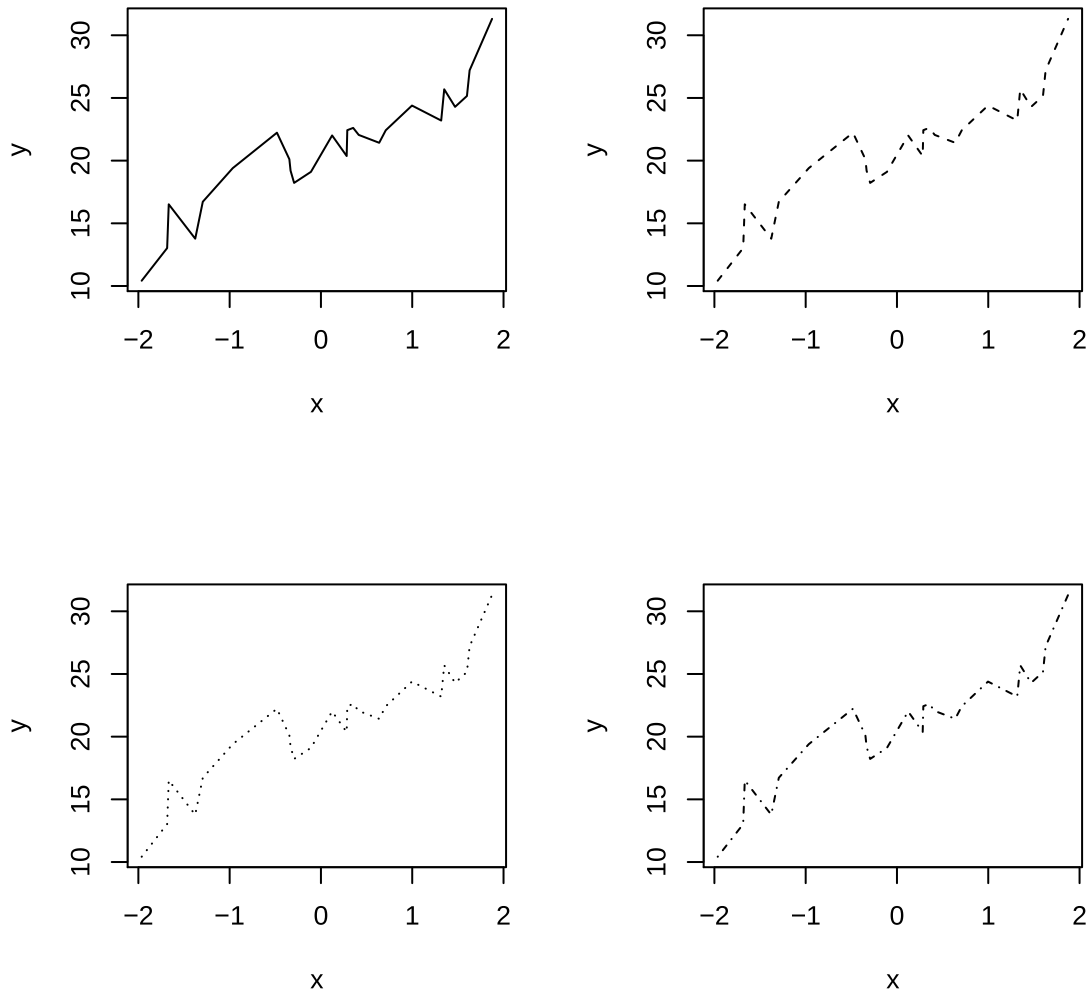


Figure 3: Alternative line types

Or the *line width* (`lwd`):

```
#explore different plot 'type'
par(mfrow=c(2,2))
plot(x,y,type='l', lty=1,lwd=1)
plot(x,y,type='l', lty=1,lwd=2)
plot(x,y,type='l', lty=1,lwd=3)
plot(x,y,type='l', lty=1,lwd=4)
```

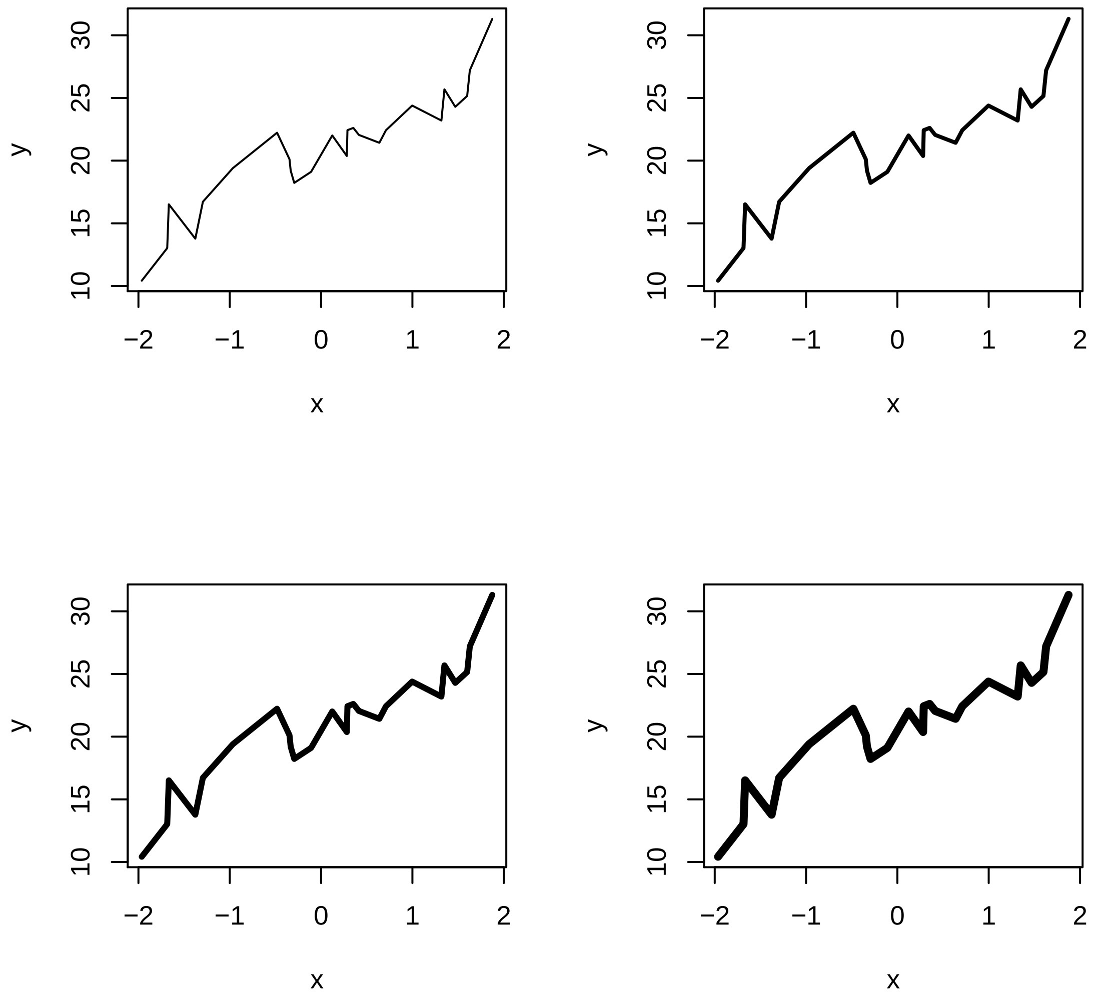


Figure 4: Alternative line widths

We can change what *symbol* (point character `pch`) we want to use:

```
#explore different plot 'type'
par(mfrow=c(2,2))
plot(x,y,type='p', pch=1)
plot(x,y,type='p', pch=15)
plot(x,y,type='p', pch=16)
plot(x,y,type='p', pch=17)
```

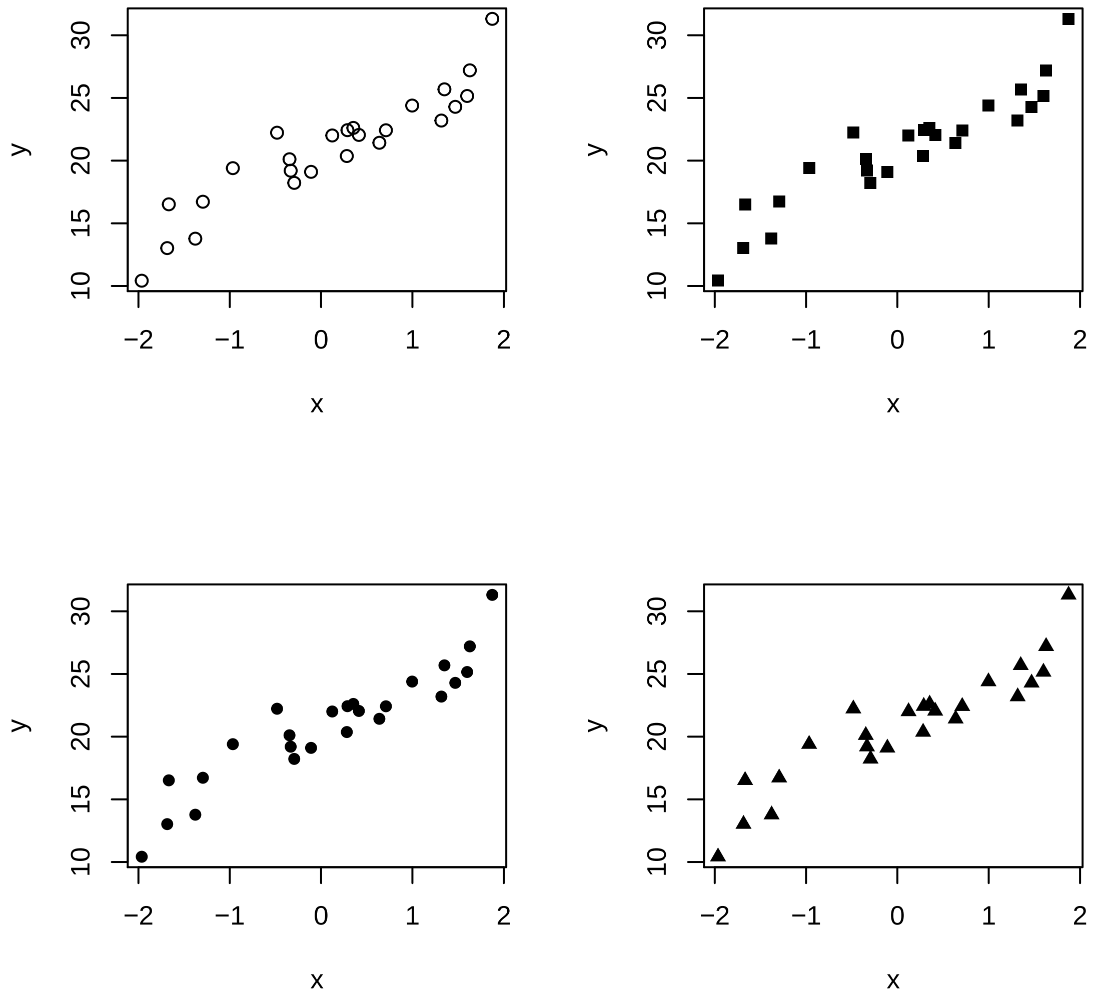


Figure 5: Alternative point symbols

Or the *size* of the symbol (character expansion `cex`):

```
#explore different plot 'type'
par(mfrow=c(2,2))
plot(x,y,type='p', pch=1, cex=0.5)
plot(x,y,type='p', pch=1, cex=1)
plot(x,y,type='p', pch=1, cex=1.5)
plot(x,y,type='p', pch=1, cex=2)
```

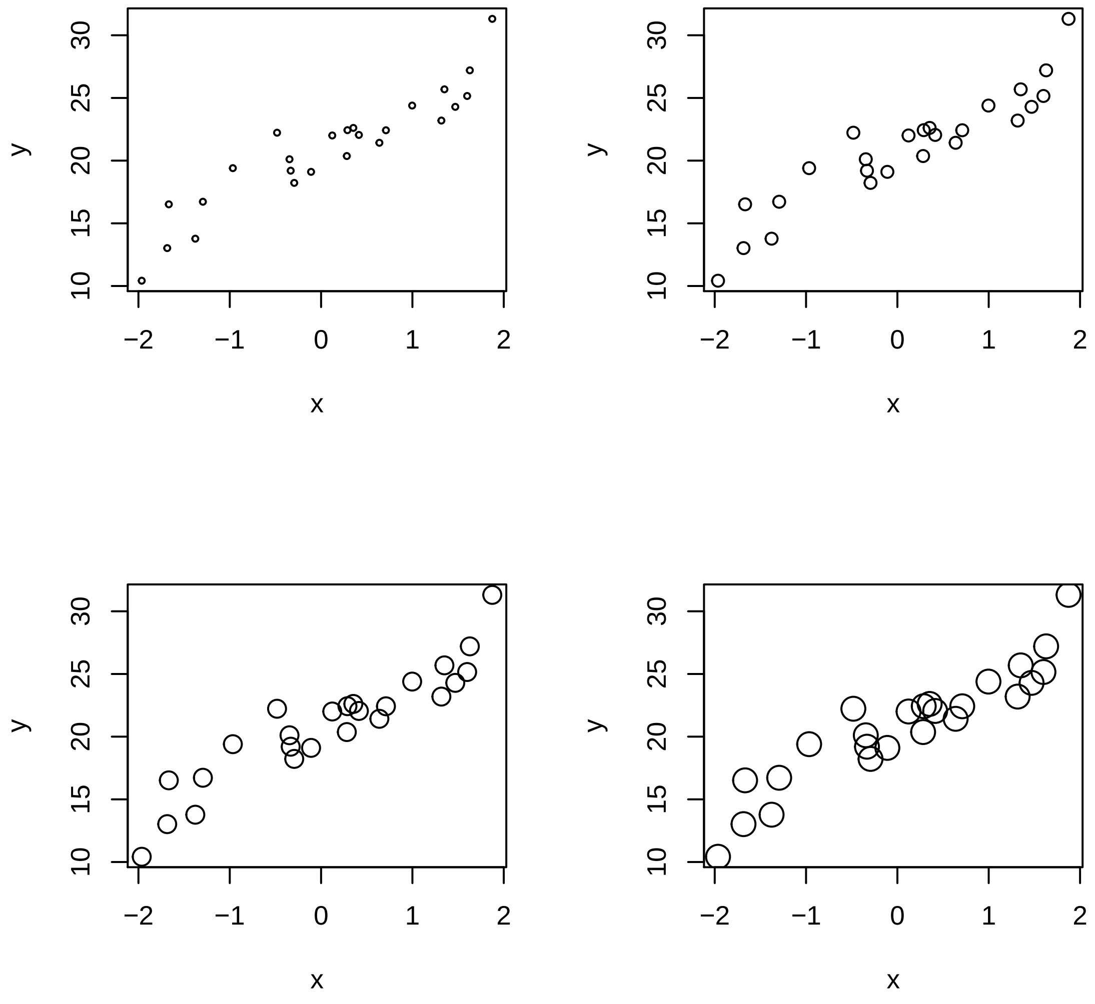


Figure 6: Alternative point sizes

We can change what *color* (`col`) we want to use for lines or poin:

```
#explore different plot 'type'
par(mfrow=c(2,2))
plot(x,y,type='l', col=1)
plot(x,y,type='l', col=2)
plot(x,y,type='p', pch=16, col=3)
plot(x,y,type='p', pch=16, col=4)
```

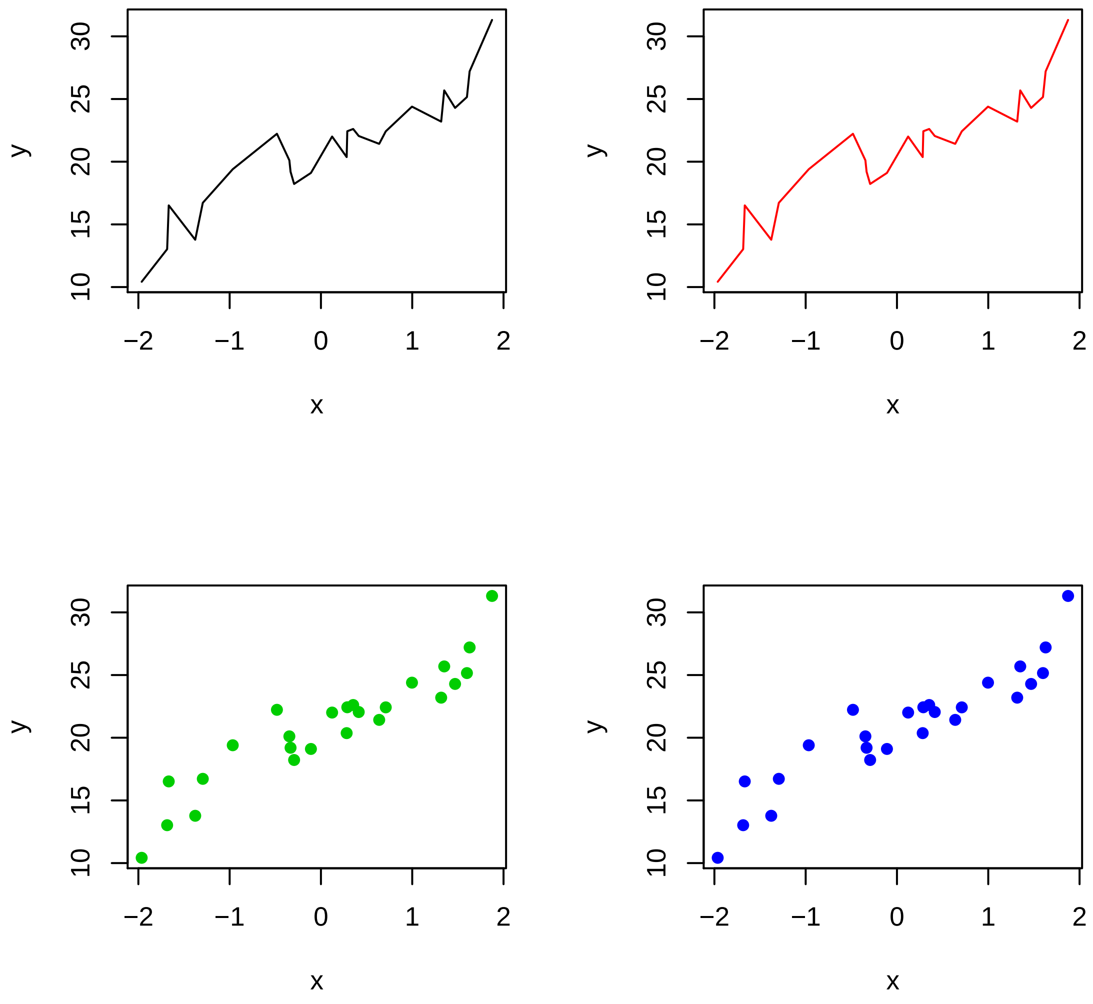


Figure 7: Alternative colors

The options are endless. To see a complete list of plot controls, look at the plotting help files, in particular `?par()` or `help(par)`. Most or all of the par commands to control the graphics can be given as arguments to the plot function (as above). However, it is also possible to set these graphics controls for the graphics device being used so that all plots to that device will adopt these same controls. This is done by passing the `par()` function before a plotting anything. Some useful examples are:

```
# partition the plotting reagion into 2 rows and 3 columns.
par(mfrow=c(2,3))

# used to overlay different plot types
par(new=TRUE)

# specifies margin size in inches (bottom, left, top, right) etc.
par(mai=c(0.6,0.5,0.1,0.5))

# all of the above in a single command
par(mfrow=c(2,3), new=TRUE, mai=c(0.6,0.5,0.1,0.5))
```

And again, there are many more options. Once a `par()` command is given, the graphics controls given by the function are set for the current graphics device until changed by a subsequent `par()` command or by changing the controls within a plot function.

Of course there are many different kinds of plots for displaying data. The basic `plot()` function is simply a starting point. There are many different so-called *high-level* plotting functions, for example:

Plotting function	Description
`plot()`	*line/point plots, output depends on data class*
`hist()`	*histogram of single variable*
`boxplot()`	*box-and-whisker plot of single variable*
`qqnorm()`	*quantile-quantile plot of single variable*
`coplot()`	*graphs of 3 or more variables*
`image()`	*draws grid of rectangles using 3 variables*
`contour()`	*draws contours using 3 variables*
`persp()`	*draws 3D surface*
`pairs()`	*all pairwise plots between multiple variables*

In addition, there are many so-called *low-level* plotting functions used to plot additional elements over an existing plot (i.e., overlays). These low-level functions are always called after a high-level command in order to supplement the high-level plot. Some examples of low-level commands include:

Plotting function	Description
`points()`	*add points to an existing plot*
`lines()`	*add lines to an existing plot*
`text()`	*add text to an existing plot*
`abline()`	*draw a line in intercept and slope form across an existing plot*
`polygon()`	*draw a polyon on an existing plot*
`legend()`	*add a legend to an existing plot*
`title()`	*add a title to an existing plot*
`axis()`	*add further axis scales to an existing plot*

There is of course much more detail to plotting in `R`, but this should suffice for now. We will be making extensive use of the plotting capabilities of `R` throughout this course.

16 Getting Help in `R`

There are lots of ways to get help in `R`. You can access information about any function you have installed by going to the help tab in **R Studio** and typing in the function name in the search window. Alternatively, you can use the `help()` function by adding the function name inside the parantheses, or, as we did previously, using the question mark:

```
help(plot)
?lines()
```

You can also search the help files for keywords using the `help.search()` function or by using double question marks:

```
help.search('mahalanobis distance')
#or
??'mahalanobis distance'
```

Note, the quotes around the phrase are necessary because it contains spaces. For a single key work search, the quotes are unnecessary. To search the **R-project** listserve for information on a particular function or, more importantly, on a topic or keyword, type:

```r
RSiteSearch("mahalanobis distance")
```

This will conduct a targeted search of documented `R` functions, documents and the help list. This is a great way to see what other people have done to solve a particular problem.

Lastly, Google or any other search engine will almost always be able to provide you with the answere you need (eventually!). To narrow down the search to `R` related results add *CRAN*, or *R software*, or *R* to your keyword, e.g.,: *cran mahalanobis distance*

17 Libraries and Packages

While `R` is an expansive language with a large number of routines already included, it doesn't include everything. Fortunately, the core functions are easily augmented with additional user-written functions, combined and published in what are called 'packages'. Accordingly, it's necessary to know how to load packages to make the most of `R`. In **R Studio**, click on the Packages tab in the lower right window. This will list all the packages you have installed with `R`. The ones that are checked, mean they are available for use in your current `R` session. If you try to access a function from a package that has not yet been loaded into your current `R` session you will get an error. Therefore, it is a good idea to load all the packages you might need for a given `R` session up front. For example, if we know we'll be using functions from the **MASS** package, simply type either of the following:

```r
library(MASS)
require(MASS)
```

If the library you want is not installed, you will have to install it yourself. **R Studio** makes this pretty easy. Go to the packages tab and click the *Install* button. In this window, you'll want to choose the *Repository (CRAN)* option for Install from. Then, start typing in the name of a package. Let's install the **vegan** package. Type **vegan** into the Packages box. You can accept the rest of the defaults and click the Install button. This will tell `R` to install this package and you'll see the install progress in your console. NOTE: you need to be connected to the internet to install any packages located on CRAN servers.

18 R Workspaces

When we open R, we also open a workspace. Any libraries that we load or objects that we create are stored in that workspace, which is really nothing more than named computer memory. When we are done with a session, it may be convenient to save the workspace so that we pick up right were we left off at some later time. R allows us to save the current workspace and load previously saved workspaces. An R workspace is given a user specified file name with an .RData extension. This is most useful if the session involved lots of processing to create temporary files (i.e., not save to disk) and we don't want to have to repeat that processing the next time we are ready to continue working. So, we simply save the workspace using the file drop-down menu and the next time we are ready to continue working on this particular project, we simply load the saved workspace and pick up right where we left off. If the processing time to get back to the point of interest is very short, we may not need to save the workspace and instead simply rerun the script back to the point of interest.

19 Tutorials for Learning R

A new set of teaching modules for using the R programming environment in ecology and epidemiology is available through the open-access on-line peer-reviewed journal The Plant Health Instructor (PHI, http://www.apsnet.org/education/AdvancedPlantPath/Topics/RModules/default.html). You might explore these modules on your own to better prepare yourself for your future life in R.

20 Local R user listserve

There is an internal (for ECO, OEB, etc.) R users group that will, hopefully, provide a friendly and safe environment for asking R related questions. As you continue to develop your proficiency in R you will realize that the environment provides a great deal of freedom and flexibility for handling, manipulating, visualizing and analyzing data. If you are struggling with some coding, it is *highly* likely that the problem you are trying to solve has been closely approximated, or even solved, by someone in the department. The point of this user group is to provide a space for you to ask questions about how to do such things so you can spend time on research, rather than coding.

- to subscribe, visit: https://list.umass.edu/mailman/listinfo/rusers

- you will be added within a few days
- only subscribers can email the group
- rusers@eco.umass.edu

NB: This is **NOT** a statistics forum. Students should use the *QSG* consulting sessions to seek statistical advice. This is meant to alleviate some of the frustrations of getting `R` to do precisely what you it to do, without taking time away from your core research.