Goals

This stand alone tutorial aims to familiarize students in the environmental sciences with introductory use of the open-source statistical software package R. Learning R requires some activation energy, but once the basic skills are in place the payoff is great; complex analyses can be performed quickly, easily and reproducibly. The goal of this tutorial is to lower that activation energy by combining limited discussion with example code for some basic statistical analyses that are common in ecology. For our purposes, it is important to focus on learning the programming language rather than trying to learn statistics and coding simultaneously.

This tutorial takes a different strategy than other R tutorials. In my experience learning R, I've found that many tutorials are simply too intimidating for a first time user, or too sparse to get much useful information (e.g. help files associated with particular functions), or too verbose - requiring a hundred pages of reading before getting to something like regression. Here I try to strike a balance between a concise explanation of models/code and sufficient detail to explore models and provide research-ready skills. To achieve this, I provide many examples for each task to make the pattern recognition in the code easier. It is critical that you actually enter the code into the R console for the examples I provide; reading through the text alone will not be helpful, particularly when faced with the exercise at the end of each chapter. The best strategy is to type the code yourself rather than copying and pasting; there are only a few commands longer than two lines where retyping slows you down.

Ultimately, these examples should prepare you to explore further on your own and learn more advanced analyses. You'll only learn R through lots trial and error so try not to be discouraged when you see error messages or code looks intimidating and complex. The best way to deal with this is exploration. This tutorial is designed with the expectation that you'll explore as you go; be sure to pause periodically and see if you understand how the functions work or whether modifying the code slightly produces the expected results. I'll provide some explicit suggestions for this in section 1.3.

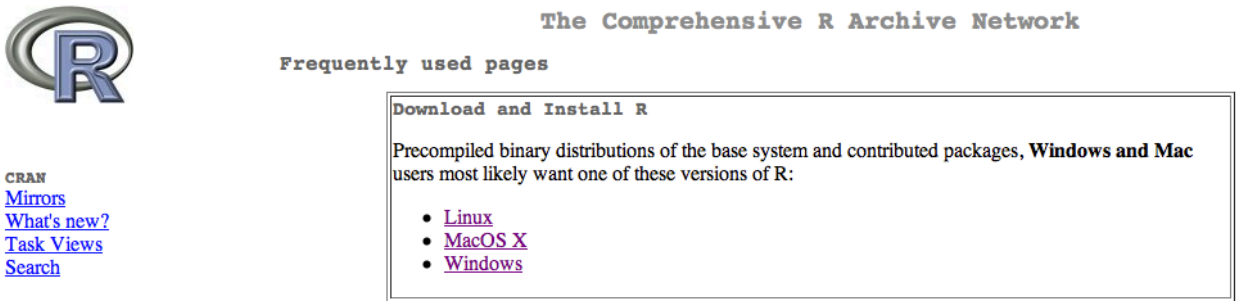
Click the page number below to jump to the associated point in the document

Table of Contents

0. Getting started

0.1 What is R?

R is an open source statistical programming language with a command line user interface that provides tremendous flexibility to customize code or perform complex analyses with code built by others. You can download the software here http://cran.mirrors.hoobly.com/. Click on your operating system as shown in the screen shot below.

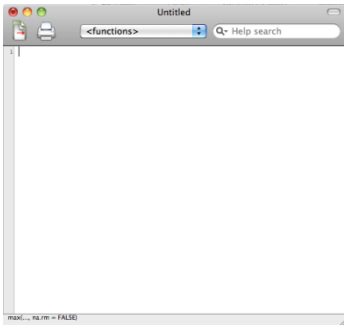


On the next page, choose the link for R-2.12.1.pkg (latest version) for Macs or base for Windows. I'll assume you can take it from there.

Next, some jargon. The window that opens when you start R is usually called the *console*. When you open R, you start a new *session*. All the variables, functions, data, etc. that R stores during a session are part of your *workspace.* You enter your code on the *command line*. All your files are stored in your *working directory.* This is just a bunch of jargon to keep normal people away from programmers.

0.2 Entering code

R uses a command line to tell it what to do. You can type directly on the command line, but it is usually better to save code in a text file to modify or rerun analyses easily. This is far superior to push-button programs where undoing and redoing operations can be difficult or hard to recall. This command line interface allows users to share their code, so once something has been written, anyone can use it. Also, it's becoming popular to include R code in the appendices of published manuscripts, which considerably extends the number and types of analyses that one can easily construct. Throughout this tutorial I provide code that can be copied and pasted on the command line to run. However, when writing your own code, its best to store it in text files using a standalone text editor (e.g. Notepad (PC) or Textedit (Mac)). R has a built in text editor but it's essentially terrible. However, you may find it easiest to just get started with it, for the sake of this tutorial. You can open the editor by going to the *File* menu and selecting *New* (Mac) or *New Script* (PC). A new window will open that looks something like the one shown, depending on whether you're using a Mac (like me) or a PC. From the text editor you can highlight code to copy and paste it to the console to run it. But some nice keyboard shortcuts are possible. On a Mac the default shortcuts are as follows: you can highlight code and use *command + return* to send it to R. Hitting *command + return* without highlighting code will send just the line that your cursor is on. On a PC you can use *ctrl+R* to send a line or highlighted code to R. I recommend typing all the code in this tutorial into the editor as you go. Then, if (when) you make mistakes you can modify the code in the editor rather than dealing with R's command line. Also, if



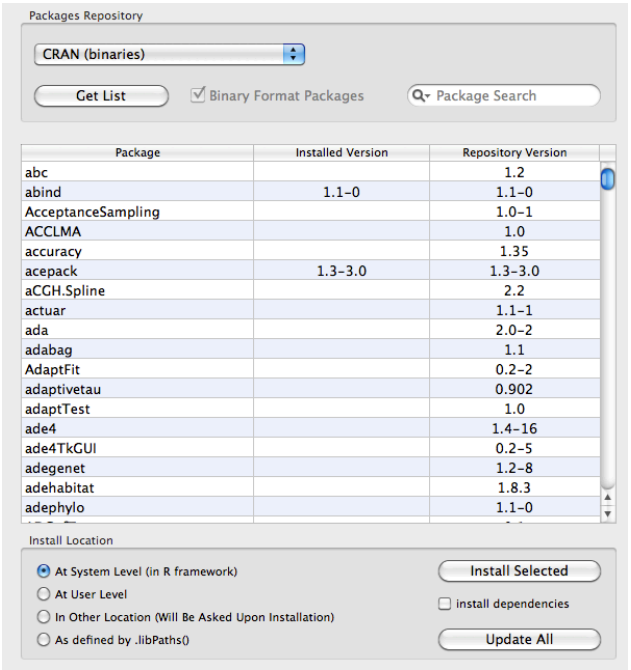
you don't finish the tutorial in one sitting, you can re-enter portions of code that you typed earlier easily. You can save this file by choosing *Save* from the *File* menu. You'll also notice that the editor has a couple features, at least on a Mac, to make it easier to look at:

1. Different parts of code, like functions, numbers, strings and comments are shown in different colors to make it easier to look at.
2. It will autocomplete brackets, parentheses and quotes for you (every time you use an open quote, you're guaranteed to need a close quote, so this just puts it in there for you).
3. When you type a function name, at the bottom of the editor a little helper appears that indicates the names and syntax of the arguments (this statement will make more sense after section 1.3).

These options can be set in most 3rd party text editors too (like Tinn-R and TextWrangler). But again, virtually all other text editors ever conceived are better than the built in editor. Many windows users seem to like Tinn-R (http://www.sciviews.org/Tinn-R/), while Textwrangler (http://www.barebones.com/products/textwrangler/) works well for Macs. Emacs and its derivatives are another type of text editor that provides the most flexible interface but it's rather challenging to get used to and probably should be avoided until you're a pretty experienced R user. All these editors are nice because they're designed with programmers in mind and allow you to customize the features mentioned above, open multiple sessions of R, and view multiple documents easily, all while running much faster (try opening 4-5 documents with the R-editor and you'll see what I mean).

0.3 R libraries

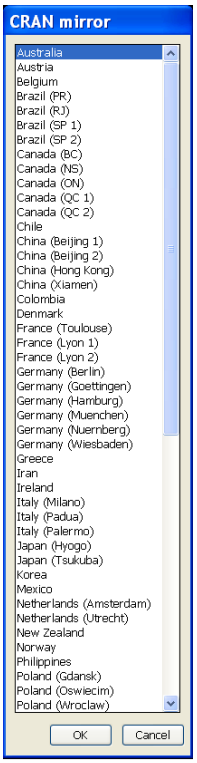
All code available in R is contributed by users (=open source). For almost any analysis you can imagine, user contributed code is available by installing *packages* or *libraries*. Packages contain suites of related functions; you'll often find many different packages that do the same thing, so exploring the options to find the best way to build models is usually valuable.



On a Mac:
You can see and download the available packages using the *Packages and Data* menu at the top of your screen. Select the *Package Installer* and the screen shown on the right will appear. Choose *Get List* to see all the available packages. A list of locations will appear where you can download from; choose a US based location. Next, you can select the packages you'd like to install; always be sure to check *install dependencies* to ensure that you'll have all the functions on which your selected packages depend. Leave all other options at default values. For this tutorial, you should install vioplot, hdrcde, Hmisc, vegan and plotrix. You'll have to wait a moment while these are downloading; it may appear like R is ignoring you, but if you've got a spinning wheel in the top right corner of your R window, that means its processing.

On a PC:

You can see and download the available packages using the *Packages* menu at the top of your screen. Select *Install packages* on a PC and the screen shown on the next page will appear. Choose a US based location to download from. Next, you can select the packages you'd like to install. For this tutorial, you should install vioplot, hdrcde, Hmisc, vegan and plotrix. (Press *ctrl* while clicking to select multiple packages.) You'll have to wait a moment while these are downloading.



Once these are installed, you must load the package before the functions become available to you. R tries to keep the minimum amount of stuff in its working memory, so you need to load these each time you open R (if you plan to use them). To load these packages, copy the following code to the command line:

```
library(hdrcde)
library(vioplot)
library(plotrix)
library(Hmisc)
library(vegan)
```

After loading hdrcde you'll see the following to indicate that some related packages are also being loaded, and similar such stuff for the other three packages.

```
Loading required package: locfit
Loading required package: akima
Loading required package: lattice
locfit 1.5-6   2010-01-20

Attaching package: 'locfit'

The following object(s) are masked _by_ '.GlobalEnv':

    expit

Loading required package: ash
Loading required package: ks
Loading required package: KernSmooth
KernSmooth 2.23 loaded
Copyright M. P. Wand 1997-2009
Loading required package: mvtnorm
hdrcde 2.15 loaded
```

Note that if you close R before finishing this tutorial, you'll need to reload these packages using `library()` as shown above (but you will not have to download them again).

1. Basic syntax

1.1 Entering and Referencing Data

R allows you to enter and evaluate mathematical expressions in same way that you might have done before on graphing calculator. One can simply type in an expression and hit enter. Note that you don't need an '=' like in Excel. For example,

```
2+3
#[1] 5

2*(456-79)
#[1] 754
```

In this tutorial, I denote the output of each calculation with #; I'll use this throughout the tutorial when it's helpful to show output. Note that I also don't show the '>' symbol at the beginning of each input line like you see on the console and in some tutorials; you'll find that this makes copying and pasting my code a lot easier.

You can assign numeric values names using an equals sign for easy reference later. For example, the following will create a variable named "num" and assign it the value of 5, which can then be used in later code:

```
num=2+3
5*num
#[1] 25
```

A variable name can also be assigned to a vector of numbers. This can be convenient when you want to perform the same operation on a group of numbers. Below, `f` is a variable name, and `c` is an example of a function, which in this case concatenates a group of numbers to create a vector.

```
f=c(4,5,6,7,8,9,10,11)
10*f
#[1]  40  50  60  70  80  90 100 110
```

More complex operations can be performed with different mathematical functions.

```
exp(f)
log(f)
sin(f)
f^2
sqrt(f)
```

When fixing mistakes, it's also useful to note, that if your cursor is on the command line and you press the up arrow on the keyboard you get the code that you typed on the previous line. Hitting up multiple times brings up earlier lines. (If you have any experience with Unix or Matlab, you might also hope that typing a few letters and hitting Tab would autocomplete function names or search earlier lines but unfortunately this isn't the case.) When editing code it can also be useful to omit certain lines that you're working on, which can be done using the # symbol. Everything that follows a # on the same line is 'commented out', meaning that R doesn't try to evaluate it. This phrase comes

from that fact that you can add comments to annotate your code by simply using a # at the beginning of the line. I'll use this strategy throughout the tutorial to indicate what certain lines of code do by using a double # (##). I'll also comment out the output of code throughout this tutorial using a single # so that it doesn't cause errors if it's pasted in to the R console along with other code. Run the next section of code to see how it works.

```
##compute sum of a vector
f=c(4,5,6,7,8,9,10,11) ##define vector
sum(f) #evaluate sum
#[1] 60
```

One can write vector succinctly using `seq` (for 'sequence').

```
f=seq(from=4,to=11,by=1)
f
#[1]  4  5  6  7  8  9 10 11
```

or even more succinctly as

```
f=seq(4,11,1)
f
#[1]  4  5  6  7  8  9 10 11
```

or even more succinctly, if the values are only integers

```
f=4:11
```

Here are a few more examples to demonstrate the flexibility of `seq` and a similar function, `rep`.

```
seq(10)
#[1]  1  2  3  4  5  6  7  8  9 10
seq(0,1,length=10)
#[1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667
#[8] 0.7777778 0.8888889 1.0000000
seq(0,1,by=0.1)
#[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
rep(1,3)
#[1] 1 1 1
c(rep(1,3),rep(2,2),rep(-1,4))
#[1]  1  1  1  2  2 -1 -1 -1 -1
```

It's often nice, once you become familiar with a function's arguments, to specify them without the labels. The only catch is that they have to be specified in the right order if you omit the labels (i.e. the value for 'from', followed by the value for 'to' followed by the value for 'by').

It's typically necessary when working with data to extract subsets from it. There are many different ways to perform the same *subsetting*, or the process of choosing just a portion of your data; I'll describe a few. These methods are all useful in different contexts, but you may find it easiest to choose your favorite method and just stick with it to start. Say that we need the third value of the vector `f` that we created above. `f` is an example an *object* in R; objects are the quantities on which R f= performs operations. I'll demonstrate other types of objects shortly. Anyhow, to specify the third

value of `f`, we use square brackets.

```
f[3]
# [1] 6
```

If instead we want the first four values, we'd use

```
f[1:4]
#[1] 4 5 6 7
```

where `:` stands for the sequence of integers between 1 and 4. Or for just the first, fourth and sixth elements:

```
f[c(1,4,6)]
# [1] 4 7 9
```

Referencing becomes a little more complicated with fancier objects such as matrices, which have both rows and columns. To demonstrate, we'll use one of the built in data sets from R called `iris`, which contains trait data for a group of iris species. Type `iris` to see what this data set looks like. To extract a value from a matrix, you must specify the row(s) and column(s), in that order, separated by commas. The following should be sufficient to demonstrate the options:

```
iris[1,2]
iris[1:10,1:3]
iris[c(1,3),c(1,3,5)]
iris[,1]
```

Note that you only need the `c` function above when you're referencing nonconsecutive numbers. For example, on the second line we could have written `1:10` as `c(1,2,3,4,5,6,7,8,9,10),` but `1:10` is easier. You can also omit elements using a minus sign.

```
x=iris[1:10,1]
x[-c(1,3,5)]
```

Note that if you don't specify row numbers, as in the first line below (there's no numbers before the comma, where a row number should be), you get back all the rows. If the columns of your matrix have names, they can also be subsetted by referring to them as follows. Note the equivalence of the two commands below.

```
iris[,1]
iris$Sepal.Length
```

The $ in the last line tells R that you want a certain column of the matrix. You could read this as 'iris, column, sepal length'. It can also be useful to embed logical operators (discussed more in section 2.1) when subsetting. The following two equivalent commands extract the data for a single species, setosa. Here, == is the operator that determine if the values in the fifth column of iris have the value 'setosa'.

```
iris[iris[,5]=='setosa',]
iris[iris$Species=='setosa',]
```

The next two lines ask for samples with large sepal lengths using the *greater than* (>) operator and

the *greater than or equal* (>=) operator. The square brackets here are equivalent to saying "where"; in prose the first expression says: return the rows of the dataset iris where the value in column 1 is greater than 7.2.

```
iris[iris[,1]>7.2,]
iris[iris[,1]>=7.2,]
```

Here's a complete list of logical operators:

- > for "greater than"
- >= for "greater than or equal to"
- < for "less than"
- <= for "less than or equal to"
- == for "equal to"
- != for "not equal to"
- & for "and"
- | for "or"

Finally, when you have a large data set that you don't need to see every value for, but you want to know what's in an object, use `head` to get the first 6 rows.

```
head(iris)
```

It is also possible to create larger dimensional objects to store data, called arrays, but I want to advise you at the outset that this is usually a bad idea. Say you have a 20 x 20 matrix where the rows and columns correspond to longitude and latitude. This is a 2 dimensional matrix that can store the values for some variable that changes in space, like temperature. But what if you have these values for temperature at three different times? You could create a 20 x 20 x 3 array to store this information, which amounts to stacking these matrices on top of one another. But this will usually cause you aggravation when you try to subset the data, so it's best to develop good practices from the start. This data set should instead be written as a matrix and could have five columns containing the longitude, latitude, temperature at time 1, temperature at time 2, and temperature at time 3. I'll demonstrate how to combine such matrices in this manner in section 1.4 Loops.

Another important type of object is a list. These are often the output of functions. Lists store multiple dissimilar objects together. We can combine the following three objects in a list called a:

```
a=list(b=f[1:3],c=TRUE, d=iris[1:3,1:3])
```

The elements of `a` can be recovered in two equivalent ways

```
## for the first element of the list
a[[1]]
a$b
## for the second element of the list
a[[2]]
a$c
```

You can also include multiple operations in the same command, for example to extract the first row of `d` in list `a`.

```
a[[3]][1,]
a$d[1,]
```

1.2 Data Classes

One of the greatest challenges for new R users is to get their data in to a useful format for their desired analysis. This is going to frustrate you. Plan for it. But once you have the hang of it, the payoffs are enormous because exploring your data with a variety of types of analysis is straightforward.

Next, I'll introduce a few of the common data classes– numeric, matrix, character, logical, and data frame. One can query the class of data using `class()`. First are two types of numeric data, a single numeric element and a vector of numeric elements. Below, the function `c` is used to combine data in the form of a vector and is quite commonly used.

```
f=4.4
class(f)

f=c(4,5,6,7,8,9,10,11)
class(f)
```

Continuing with the demonstration of different data types, we'll use `seq` in combination with `cbind` to make a matrix.

```
f=cbind(seq(1,4,1),seq(3,4,.33),seq(10,40,10))
class(f)
```

Note that `cbind` (for 'column bind') combines vectors (i.e. lists of numbers) as columns of a matrix. An analogous function `rbind` combines vectors as rows. Alternatively, you could create a matrix by specifying a list of numbers and the number of rows in which to distribute them.

```
matrix(1:20,nrow=5)
```

Next there are character and logical data:

```
f='tall'
class(f)
f=TRUE
class(f)
```

Note that logical data denotes TRUE/FALSE with all upper case, while character data is any other string of characters enclosed in quotes.

Any of these data classes can be combined in a data frame. Each column of a data frame can correspond to a different class. In creating the data frame below, note the use of `sample` to take a random sample from a vector. The first argument specifies the values from which to sample, the second (`size`) determines the number of samples and the third (`replace`) says whether a number can be sampled more than once. The vectors combined in this way must have the same lengths.

```
age=sample(1:20,size=30,replace=TRUE)
height=sample(seq(40,60,.25),30,replace=TRUE)
```

```
reproduction=sample(c('none','flower','seed'),30,replace=TRUE)
predation=sample(c(TRUE,FALSE),30,replace=TRUE)
f=data.frame(age,height,reproduction,predation)
```

Note that each column has retained its respective data class except for `reproduction`. The three types of characters in `reproduction` are interpreted as different levels of a factor. A factor is simply a categorical variable.

```
class(f[,1])
class(f[,2])
class(f[,3])
class(f[,4])
```

A few hangups are common. First, only certain data classes work with certain functions. For example you can't do algebra on factors. Second, there is a difference between a `matrix` and a `data.frame` even though both store information in rows and columns. A matrix can store data classes of only one type. For example, by combining a numeric vector and a factor into a matrix, all elements are converted to factors.

```
f=cbind(height,reproduction)
class(f)
class(f[,1])
class(f[,2])
```

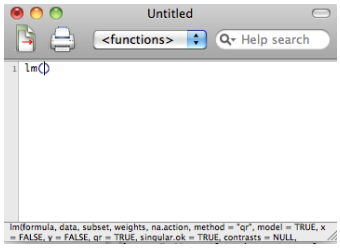
As another example, here's another way to construct and query a data frame.

```
mydata <- data.frame(y=c(1.2,3.6,5.1,4.2,2.1),
x1=c(1.5,2.5,6,3.1,2.2),x2=factor(c(1,1,1,2,2)))
mydata
#    y  x1 x2
# 1 1.2 1.5  1
# 2 3.6 2.5  1
# 3 5.1 6.0  1
# 4 4.2 3.1  2
# 5 2.1 2.2  2
mydata$y
# [1] 1.2 3.6 5.1 4.2 2.1
dimnames(mydata)
# [[1]]
# [1] "1" "2" "3" "4" "5"

# [[2]]
# [1] "y"  "x1" "x2"

dimnames(mydata)[[2]]
# [1] "y"  "x1" "x2"
```

1.3 Help!



You've probably already found yourself wondering what's happening when you enter certain bits of the code above. What should you do? Often the first issue you face is typos: putting a square bracket where there should be a parenthesis or forgetting a comma. The only way to avoid this is practice, which is why I encourage you to type the code rather than copying and pasting. Next, some of the lines of code may seem confusing. You should try making minor changes and seeing if you get the desired output. This sort of tinkering is the way most people learn R. Next, if you're on a Mac and using a function, you'll notice that, once you type its name in the R text editor or in the console, the name of that function its arguments are displayed along the bottom line of whichever one you're typing in (see figure ->). If you need more info, you should look in the function's help file.

All functions in R have help files associated with them. If you know the name of the function, typing `?functionname` brings up the file. If you don't know the name type `??` immediately followed (no space) by the beginning of the file. For more complicated searches involving multiple vague terms, the help box at the top of the GUI (Mac) or *Help* in the menu bar (PC) is easiest. However, R's help is not great if you're searching around for a topic, rather than a function you can't recall; a Google search will be much more helpful.

Help files have a common layout. The content varies depending on whether the contributors of the package felt like being helpful. Functions that are part of the built in packages or which are widely used by a variety of packages are pretty well documented. But obscure packages with only a few specific functions can be a pain, particularly when you're just learning R. The help file layouts are pretty self explanatory, but note the following:
Arguments – describe all the arguments and the rules for specifying
Details – description of how one might use the function; highly variable in quality
Values – these are output in a list and can be called as shown in the end of section 1.2
Examples – At the end of the file and highly variable in quality. I often find that running the example code is easier than wading through the help file text.

1.4 Built-in Functions

The strength of R is in its vast library of user-contributed functions, which are capable of performing almost any statistical operation you can imagine. These functions operate on the same principle as the mathematical functions shown in the previous section; the function performs some sort of operation, say fitting a linear regression, based on the data you feed it and the settings you specify. Choosing a function determines the operation you want to perform, and this function needs to be told what data to use and the details of how you want it done. The data and settings are called 'arguments'. Functions always begin with the name of the function followed by the arguments enclosed in parentheses and separated by commas, which can be generally written as:

function(data,setting1=A,setting2=B,setting3=C).

We'll begin with a few basic functions that are helpful when referencing data, as discussed in section 1.1.

```
x=c(1,3,2,10,5)
```

```
sort(x)                 ## increasing order
#[1]  1  2  3  5 10
sort(x, decreasing=T)  ## decreasing order
#[1] 10  5  3  2  1
length(x)               ## number of elements in x
#[1] 5
```

Next we construct a matrix.

```
x=1:12
matrix(x,nrow=3,ncol=4)
```

This tells the matrix to take the data, which is the sequence of integers from 1 to 12, and construct a matrix with 3 rows and 4 columns. Most functions have default values for all their settings, which simplifies your code because you only need to specify the settings that you want to modify. In `matrix`, a setting called `byrow` can be set to `TRUE` or `FALSE`, but the default is `TRUE`. This tells `matrix` to fill the first row first, followed by the second row, etc. Setting `byrow=FALSE` fills column 1 first, followed by column 2, etc. There is often a great deal of flexibility in many functions that can be nice to recall, but you should avoid shortcuts until you're familiar with a particular function. For example, if you type the values of the arguments in the default order that they're fed to the function, you don't need to specify the setting names. The following operation provides the same matrix as above. Use this sort of shortcut at your own discretion.

```
matrix(1:12,3,4)
```

Note that if you mix up the order of these arguments, you might get an error, but sometimes, if the stars align, you can get a nonsense result, so its always important to check that the objects you're working with look like what you expect.

```
matrix(3,4,1:12) ## no error, but arguments mixed up
```

The settings can be specified in any order you please as long as you provide their names. Again, we get the same matrix from the following

```
matrix(1:12,ncol=4,nrow=3)
```

Many functions also allow 'partial matching' of arguments and their values, which means that you can just give the first few letters of the argument as long as they avoid ambiguity.

```
matrix(1:12,nc=4,nr=3)
```

As another example, let's consider the function `lm`, which constructs a linear regression model, to demonstrate a few more features. The first argument of `lm` is a formula that specifies the response variable (y), which is sepal length, and the predictors (x), which is petal length, separated by a ~. To use lm in its simplest form, type

```
lm(iris$Sepal.Length~iris$Petal.Length)
# Call:
# lm(formula = iris$Sepal.Length ~ iris$Petal.Length)

# Coefficients:
#      (Intercept)  iris$Petal.Length
#           4.3066             0.4089
```

This doesn't give much interesting info, it just reminds you of the model and gives the coefficients. It's easiest to extract all the other useful stuff from lm by assigning the output to an object, which I'll call `model`.

```
model=lm(iris$Sepal.Length~iris$Petal.Length)
model
```

Typing `model` just gives the same output. But `model` is actually a list with a great deal of additional information in a list. Most functions produce output in this way. Try the following for an example:

```
model$residuals
model$coeffiecients
model$fitted.values
```

All the output from a function can be found in the help file (type `?lm`) in the 'Values' section (see section 1.5 for help file layout).

Another suite of useful functions gives values of different probability distributions. If you're not comfortable with probability distributions, go here: http://en.wikipedia.org/wiki/Probability_distribution. Each distribution has an abbreviation (e.g. norm for the normal) that is preceded by a d for the density function (`dnorm`), p for the cumulative distribution function (`pnorm`), q for the quantile function (`qnorm`) and r for a random draw from the density function (`rnorm`). The following text is adapted from a nice introductory suite of tutorials at http://www.cyclismo.org/tutorial/R/index.html .

The first function we look at is dnorm. Given a set of values dnorm returns the height of the probability distribution at each point. If you only give the points, it assumes you want to use a mean of zero and standard deviation of one. There are options to use different values for the mean and standard deviation, as shown. I also demonstrate plotting these distributions, with more plotting details to come in chapter 3. You can enter all the lines at once or one at a time.

```
dnorm(0)
# [1] 0.3989423
dnorm(0,mean=4)
# [1] 0.0001338302
dnorm(0,mean=4,sd=10)
# [1] 0.03682701
v = c(0,1,2)
dnorm(v)
# [1] 0.39894228 0.24197072 0.05399097
x = seq(-20,20,by=.1)
y = dnorm(x)
plot(x,y)
y = dnorm(x,mean=2.5,sd=0.1)
plot(x,y)
```

The second function we examine is *pnorm*. Given a number or a list, pnorm computes the probability that a normally distributed random number will be less than that number. This function also goes by the rather ominous title of the "Cumulative Distribution Function." It accepts the same options as *dnorm*:

```
pnorm(0)
# [1] 0.5
pnorm(1)
# [1] 0.8413447
pnorm(0,mean=2)
# [1] 0.02275013
pnorm(0,mean=2,sd=3)
# [ 1] 0.2524925
v = c(0,1,2)
pnorm(v)
# [1] 0.5000000 0.8413447 0.9772499
x = seq(-20,20,by=.1)
y = pnorm(x)
plot(x,y)
y = pnorm(x,mean=3,sd=4)
plot(x,y)
```

The next function we look at is *qnorm* which is the inverse of *pnorm*. The idea behind *qnorm* is that you give it a probability, and it returns the number whose cumulative distribution matches the probability. For example, if you have a normally distributed random variable with mean zero and standard deviation one, then if you give the function a probability it returns the associated *Z*-score:

```
qnorm(0.5)
# [1] 0
qnorm(0.5,mean=1)
# [1] 1
qnorm(0.5,mean=1,sd=2)
# [1] 1
qnorm(0.5,mean=2,sd=2)
# [1] 2
qnorm(0.5,mean=2,sd=4)
# [1] 2
qnorm(0.25,mean=2,sd=2)
# [1] 0.6510205
qnorm(0.333)
# [1] -0.4316442
qnorm(0.333,sd=3)
# [1] -1.294933
qnorm(0.75,mean=5,sd=2)
# [1] 6.34898
v = c(0.1,0.3,0.75)
qnorm(v)
# [1] -1.2815516 -0.5244005  0.6744898
x = seq(0,1,by=.05)
y = qnorm(x)
plot(x,y)
y = qnorm(x,mean=3,sd=2)
plot(x,y)
y = qnorm(x,mean=3,sd=0.1)
plot(x,y)
```

The last function we examine is the *rnorm* function which can generate random numbers whose distribution is normal. The argument that you give it is the number of random numbers that you want, and it has optional arguments to specify the mean and standard deviation. Note your random numbers will look different than mine – that's what makes them random.

```
rnorm(4)
# [1]  1.2387271 -0.2323259 -1.2003081 -1.6718483
rnorm(4,mean=3)
# [1] 2.633080 3.617486 2.038861 2.601933
rnorm(4,mean=3,sd=3)
# [1] 4.580556 2.974903 4.756097 6.395894
rnorm(4,mean=3,sd=3)
# [1]  3.000852  3.714180 10.032021  3.295667
y = rnorm(200)
hist(y)
y = rnorm(200,mean=-2)
hist(y)
## to check whether the sampled quantiles line up with normal quantiles
qqnorm(y)
qqline(y)
```

Finally, here is a quick reference card with a list of commonly used R functions
http://cran.r-project.org/doc/contrib/Short-refcard.pdf

1.5 Writing your own functions

While there a variety of built-in functions, it is usually necessary to write custom functions to optimize efficiency in your code both from the perspective of computation time and readability. I'll just specify the general format here and develop more functions as necessary in the context of specific problems. Function syntax requires you to give the function a name, specify the names of the arguments, and enclose the operations of the functions in brackets. These functions can then be run like any other built in functions. This is most easily seen through examples.

```
## the logit function used in generalized linear models
logit = function(x){   log( 1/(1-x) )   }
logit(.6)
[1] 0.9162907
```

In the last line above, I'm computing the value of the function at x=.6. Hence R plugs in whatever values you give it. You could also feed it a vector.

```
logit(c(.6,.7,.8))
[1] 0.9162907 1.2039728 1.6094379
```

Another function example.

```
## to undo the logit transform
expit = function(x){   1/(1+exp(-x))    }
expit(0.4054651)
[1] 0.6
```

Note that I can put spaces, tabs, or returns between operations at no consequence. Once you run a function, you should end up with an output value. In the example above, the output of `expit(0.4054651)` is 0.6. But this output can be more complex, for example a list can be created as demonstrated in the function `twosam` shown below. The default when you write a function is to output the last value that it calculates. For example, in the logit function above, the last value it results of `log( 1/(1-x) )`. This means that if you want something other than the last value calculated, you must specify it. I show how to specify the output below, where I give the trivial example of outputting the last value (which it would do by default anyways).

```
cube=function(x){
     y=x*x*x
     return(y)
}
cube(3) ## to use another value of x
#[1] 27
```

You can specify the default values of your function arguments by setting them in the function assignment.

```
cube5=function(x=5){
     y=x*x*x
     return(y)
}
cube5() ## to use the default value of x=5
# [1] 125
```

As a more complex example, we'll write a function to perform a two sample t-test for the equivalence of the mean of two samples. We assume the two samples are of equal size (n) and drawn from distributions with the same variance. This requires computing the following test statistic:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{S_{X_1X_2} \cdot \sqrt{\frac{2}{n}}}$$

where

$$S_{X_1X_2} = \sqrt{\frac{S_{X_1}^2 + S_{X_2}^2}{2}}$$

Where X1 and X2 represent the samples and S represents the standard deviation (more details: http://en.wikipedia.org/wiki/Student's_t-test#Independent_two-sample_t-test). We'll output the t statistic and the value of the cumulative distribution function at this value. This function has more arguments and more output, returned as a list.

```
twosam = function(y1, y2) {
         n1  = length(y1); n2  = length(y2)
         yb1 = mean(y1);   yb2 = mean(y2)
         s1  = var(y1);    s2  = var(y2)
         s = ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
               ## the test statistic
         tst = (yb1 - yb2)/sqrt(s*(1/n1 + 1/n2))
               ## value of cumulative distribution function
         p.tst=pt(tst,n1+n2-2)
         return(list(tst=tst, p.tst=p.tst))
       }
```

When you type this in, you won't get any output. That's because we're just specifying a function, and not using it. We can test this function using two samples from a normal distribution with the same mean, which should yield a cumulative distribution function value (`p.tst`) below .95 (run it a few times; the answers will differ due to different random samples).

```
x1=rnorm(100)
x2=rnorm(100)
```

```
twosam(x1,x2)
# $tst
# [1] -0.5545724

# $p.tst
# [1] 0.2899065
```

The test does, however, pick up differences between the means when they exist. When x2 has a smaller mean, the t statistic will be larger and the cdf value greater than 0.95.

```
x1=rnorm(100)
x2=rnorm(100,mean=-1)
twosam(x1,x2)
# $tst
# [1] 7.988474

# $p.tst
# [1] 1
```

1.6 Loops

Loops are useful ways to specify repetitive operations compactly. The most common type is a *for loop*, which performs some operation FOR each value of its *index*. The index of the loop is specified between parentheses and the operations are specified between brackets. Loops need an index, which is traditionally called i, j, or k to operate over (but you can use any symbol that makes sense to you). We assign to the index some values (in this case a sequence), using `in`, that tells the loop to perform a series of commands with those values. This index tells R, 'when *i* takes the value 1, do the following, then when *i* takes the value 2, do it again, etc.

```
for( i in 1:5 ){  print(i^2)  }
# [1] 1
# [1] 4
# [1] 9
# [1] 16
# [1] 25
```

Note that to store the output of the loop (the stuff you see printed to the screen), I need to create an object to put it in first, which I assign the value NA (meaning not available). It's a good idea to assign such placeholder objects the value of NA so you know whether or not your subsequent commands have succeeded in storing any output there. Say we want to create a matrix of samples from a normal distribution with mean 0 and standard deviation of 1. For this we use rnorm, where the first argument specifies the number of samples, the second indicates the mean and the third indicates the standard deviation.

```
output=matrix(NA,nrow=10,ncol=4)
output ## check that it's a matrix of NAs before we run the loop
for( i in 1:4 ){
     output[,i]=rnorm(10,0,1)
}
output ## check the output
#           [,1]       [,2]        [,3]        [,4]
#  [1,]  1.8225579 -1.0555967 -1.88298647  0.22664135
#  [2,]  0.5893980 -0.7865660  0.46782189  0.39975791
#  [3,]  0.5732110  0.4519762  0.74401548  0.34917500
# [4,]  0.2877259 -1.2486698  1.71805493 -0.45787280
# [5,] -0.5237484 -1.1283136 -0.65075612  1.28295655
```

```
# [6,] -0.4872318 -0.3746510 -2.89605508 -0.03034306
# [7,]  0.8608489  2.0517065 -0.05874342 -0.15498445
# [8,]  1.5076224 -0.6593916 -0.30634240  0.52761700
# [9,]  0.9886803  0.0951733  0.51083565 -0.97144857
# [10,] -1.3569799 -1.0489771  0.79931376  0.49898598
```

Here are a few more examples of loops to help you get the idea. There are a few functions in here that I haven't discussed yet; use ? followed by the function name (without a space) if their meaning isn't apparent to you.

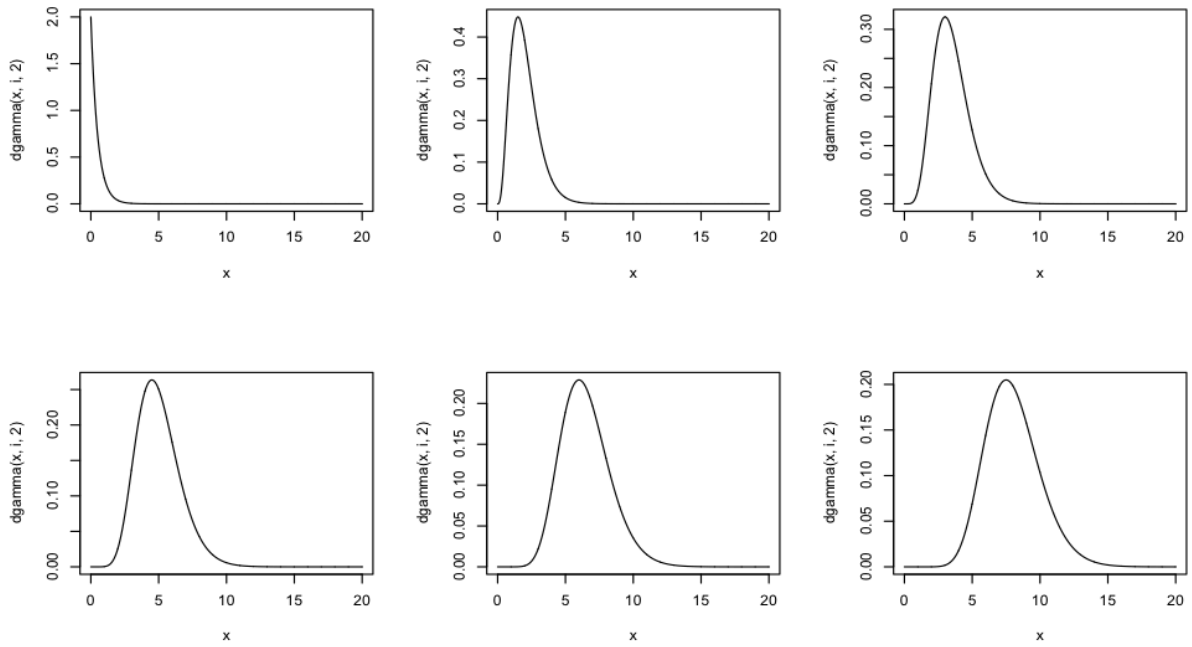
```
## calculate the mean of each of the columns from the iris data set
mean.iris=NULL
for(i in 1:4){
     mean.iris[i]=mean(iris[,i])
}
mean.iris ## print the output
# [1] 5.843333 3.057333 3.758000 1.199333


## demonstrate if/else statement
x = 1:10
z = NA
for(i in 1:length(x) ) {
     if (x[i]<5) { z[i] = x[i]-1     }
     else        { z[i] = x[i]/x[i]  }
}
z ## print the output
# [1] 0 1 2 3 1 1 1 1 1 1

## index of nonconsecutive integers
for  (ii in c(3,9,4,7)) {
     cat("The square of ",ii," is ", ii^2,".\n")
}
# The square of  3  is  9 .
# The square of  9  is  81 .
# The square of  4  is  16 .
# The square of  7  is  49 .
```

Here's another example to put multiple plots in the same figure. We'll explore what the Gamma distribution looks like for some different parameter values. Don't worry too much about the graphics commands `par` and `plot`; those will be explained in chapter 2.

```
par(mfrow=c(2,3))
x=seq(from=0,to=20,by=.01)
for( i in seq(from=1,to=16,by=3)){
     plot(x,dgamma(x,i,2), type='l') ## type=l means line
}
```



In section 1.3, I mentioned that it's always better to store data in two dimensional matrices. So imagine that you have three matrices for temperature at three different times, where the rows indicate latitude and the columns indicate longitude. First we generate the data (note the shortcut compared to the previous example)

```
T1=matrix(rnorm(80,23,2),nrow=10,ncol=8)
T2=matrix(rnorm(80,25,2.5),nrow=10,ncol=8)
T3=matrix(rnorm(80,27,3),nrow=10,ncol=8)
T1 ## see what one looks like
```

Your random numbers will look different than mine. Now we'll store this data in matrix where each row represents one of the cells of the matrices, references by its row and column number. In this new matrix the first column will give the row number, the second column gives the column number and the third-fifth columns give the temperature values from the corresponding cells for the T1, T2, T3. This requires one loop to be nested within another. The first loop denotes rows of the temperature matrices and the second denotes columns. The `counter` keeps track of the rows of the new matrix we're constructing.

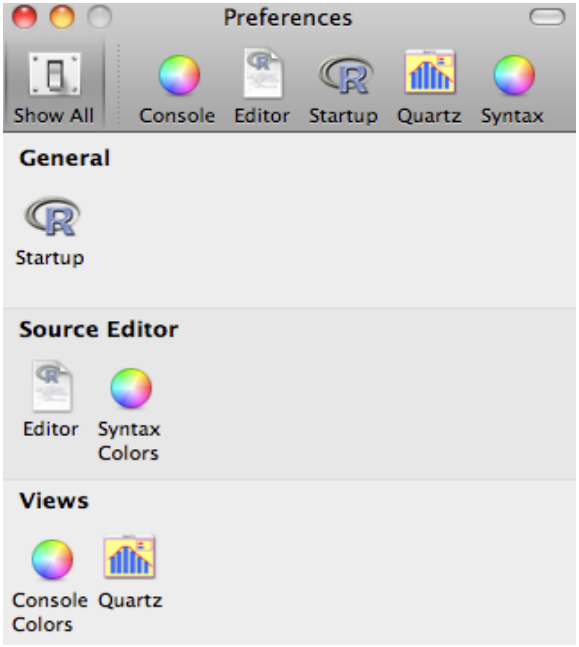
```
## note 80 rows since 80 cells in each
Tnew=data.frame(matrix(NA,nrow=80,ncol=5))
     ## specify column names
names(Tnew)=c('lat','lon','T1','T2','T3')
counter=1
for(i in 1:10){ ## row loop
     for(j in 1:8){ ## column loop
          Tnew[counter,1]=i
          Tnew[counter,2]=j
          Tnew[counter,3]=T1[i,j]
          Tnew[counter,4]=T2[i,j]
          Tnew[counter,5]=T3[i,j]
          counter=counter+1 ## to keep track of which row is filled
     } ## close column loop
} ## close row loop
```

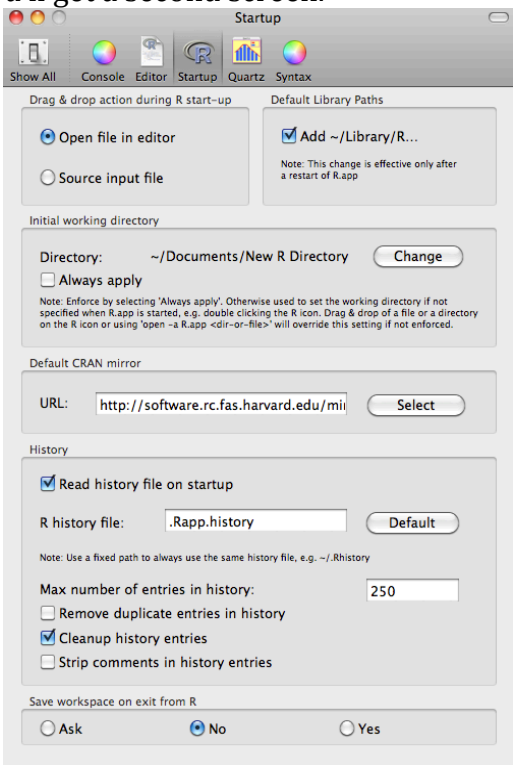
```
head(Tnew) ## look at results
#    lat lon    T1       T2       T3
# 1   1   1 21.45661 24.85239 31.10248
# 2   1   2 26.74739 22.07254 28.72009
# 3   1   3 20.27752 17.72388 30.29659
# 4   1   4 20.11895 25.68462 30.44808
# 5   1   5 22.09233 25.80475 17.70206
# 6   1   6 20.09910 24.17676 23.14202
```

1.7 Where's my stuff?

When you're reading and writing data files you should set your *working directory* – the directory where your R files will be kept. First, create a directory somewhere convenient on your computer (not using R, just do it manually). I'll call mine 'R Stuff'.

For Macs
If you are using a Mac, in order to change your working directory you need to change a default in R. Under the *R* menu (at the top of the screen) choose *Preferences*, and the following screen will appear (probably slightly different on a PC). Choose *Startup,* and you'll get a second screen.





In the second section, entitled *Initial working directory*, uncheck *Always apply*. This will allow you to change directories from the command line. You can now close the *Preferences* window.

For PCs
On a PC, the default directory is the "My Documents" folder. But if you do wish to change the default, do it by right-clicking the R icon on the desktop, going to the Shortcut tab, and changing the "Start in" folder.

Ok, back to setting up your working directory. You've already manually a folder called /R Stuff/ (using Explorer or Finder, depending on your operating system). This is where you'll keep data files

to input, or read, in to R, output from analyses, or figures that you produce. You can tell R to look for files in a particular directory using `setwd` (for 'set working directory') by specifying the full path name for the directory. On my computer (a Mac), this looks like

```
setwd("/Users/cmerow/Documents/Projects/R tutorial/R stuff/")
```

On a PC, it might look like

```
setwd("C:/Documents and Settings/Silander/My Documents/R stuff")
```

Windows users, note that you use a forward slash in R where you would use a backslash in Windows Explorer. Just be sure that you specify the whole thing, which is case sensitive. If you're unsure which directory R is currenly working in, type `getwd()`. If you're unsure where to start with specifying your path name, using `getwd()` can show you what the syntax looks like.

You may choose to organize your working directory further, for example by providing a different folder for each project. Create a folder in /R Stuff/ called /R Quickstart/. This is we're we'll store some data now.

To practice reading data in to R, we'll first simulate a data set and write it to /R Quickstart/ so we can read it back in later. First we'll consider the case where you want to save some variables you've created in R for use in your workspace. You can save these using `save` and reopen them later using `load`.

```
a=c(1,4,7)
save(a,file='R Quickstart/demo.R')
## remove a from the workspace
rm(a)
a ## check that you get an error because its gone
# Error: object 'a' not found
load('R Quickstart/demo.r')
a ## and now its back
# [1] 1 4 7
```

If you get an error relating to 'No such file or directory', that means that either you're working directory is not set as specified above or that you didn't create the subdirectory inside it called 'R Quickstart'. Also, note that I didn't have to specify the whole path name for 'R Quickstart', I just specified the part that's in a subdirectory of my working directory.

To save a group of related data, first put it in a list. Note that I write this file directly to the working directory, and not the folder 'R Quickstart'.

```
a=list(b=f[1:3],c=TRUE, d=iris[1:3,1:3])
save(a,file='demo2.RData')
rm(a)
a ## check that you get an error because its gone
# Error: object 'a' not found
load('demo2.RData')
a ## and now its back
# $b
# [1] 4 5 6
```

```
# $c
# [1] TRUE

# $d
#  Sepal.Length Sepal.Width Petal.Length
# 1          5.1         3.5          1.4
# 2          4.9         3.0          1.4
# 3          4.7         3.2          1.3
```

Saving data in this fashion makes it available to R, but what if you want to create a data file that you can open in Excel? For simple data that can be written in matrix form, R works best with .txt and .csv files. We'll use .csv files here. First, we'll generate a data frame, then save it to a .csv file.

```
a=data.frame(matrix(1:100,nr=20))
names(a)=c('a1','b1','c1','d1','e1')
write.csv(a,file='demo3.csv')
aa=read.csv('demo3.csv')
aa ## see what we get back
```

You'll notice that the matrix you saved isn't the matrix you get back. The first column of aa, labeled X, is actually the row numbers of your matrix. To avoid writing these to the file, use `row.names=FALSE`. It's also a good idea to specify whether the first row of data that you're reading contains row names, just in case there are some spaces or numbers that confuse R's attempt to guess what you intend. Our matrix has column names, so we'll use `header=TRUE`.

```
a=data.frame(matrix(1:100,nr=20))
names(a)=c('a1','b1','c1','d1','e1')
write.csv(a,file='demo4.csv',row.names=FALSE)
aa=read.csv('demo4.csv',header=TRUE)
aa ## good to go
```

2. Graphics

One of the most compelling reasons to invest in learning R is the flexibility of its graphics. I say this partly because I enjoy trying to make pretty pictures and partly because effectively communicating results can be critical to the success of research. To see some examples of basic graphics, use the following demos.

```
demos(persp)
demos(plotmath)
```

Demos are usually just eye candy, but it's worth having a look at them when you're getting started with a package because they can show the code associated the operations. To see a list of all the available demos from the packages you've installed, use

```
demo(package = .packages(all.available = TRUE))
```

Before proceeding you should check out the R graphics gallery, which contains surprisingly simple code to construct many of the figures you'll see.
http://addictedtor.free.fr/graphiques/thumbs.php

Really, go look at it before proceeding, its impressive. By copying the code, you can make plots that look as great as these with just a few edits or completely customize your plots and extract just what you like. This is also a great way to learn through examples. In this section I'll focus mostly on constructing the sorts of basic diagnostic plots that you'll need to evaluate simple models, but I'll also provide a few of the fancy plots adapted from the gallery.

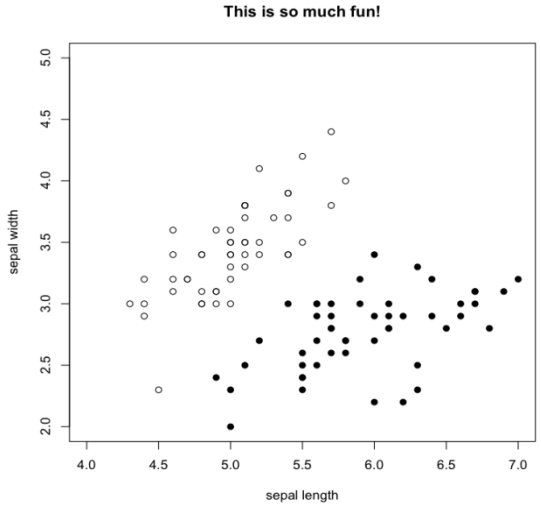
2.1 Scatter plots

I usually think in terms of regression designs, so I'll begin with scatter plots. The basic function `plot` has a plethora of options, most of which are general and are available in many other plotting functions. I'll proceed by demonstrating `plot` in detail, but keep in mind that the arguments apply to other plotting functions like `boxplot` and `hist` described below. The first two arguments of `plot` are the x and y values (as vectors), which must have the same length (otherwise it recycles values from the shorter vector and gives screwy output). I'll demonstrate with the iris data again; the argument names are fairly intuitive. Note that xlim (and ylim) set the upper and lower bounds on the axes and can be omitted if you want `plot` to figure it out for you. `main` sets the title while `xlab` and `ylab` set the axis labels. Note that `iris$Species=='setosa'` asks for the rows of the iris data set where the Species column has the value setosa. The == means 'equal to' and setosa is in quotes because it's a factor.

```
x= iris$Sepal.Length[iris$Species=='setosa']
y= iris$Sepal.Width[iris$Species=='setosa']
plot(x,y, main='This is so much fun!', xlab='sepal length', ylab='sepal
width',xlim=c(4,7),ylim=c(2,5))
```

Keep your plot window open for this next part - if you need to add points to your plot, say for a different species, this can be done as follows.

```
points(iris$Sepal.Length[iris$Species=='versicolor'],
iris$Sepal.Width[iris$Species=='versicolor'],pch=19)
```



The argument `pch` indicates the type of marker you want for the data point and is specified by a number between 1 and 255. I often forget these and refer to a function written in the help file for points (`?points`) called `pchshow` that makes a nice graphic. I include the code here; don't bother to wade through it, just run it and look at the picture.

```
pchShow <-
  function(extras = c("*",".", "o","O","0","+","-","|","%","#"),
           cex = 3, ## good for both .Device=="postscript" and "x11"
           col = "red3", bg = "gold", coltext = "brown", cextext = 1.2,
           main = paste("plot symbols :  points (...  pch = *, cex =",
                        cex,")"))
  {
    nex <- length(extras)
    np  <- 26 + nex
    ipch <- 0:(np-1)
    k <- floor(sqrt(np))
    dd <- c(-1,1)/2
    rx <- dd + range(ix <- ipch %/% k)
    ry <- dd + range(iy <- 3 + (k-1)- ipch %% k)
    pch <- as.list(ipch) # list with integers & strings
    if(nex > 0) pch[26+ 1:nex] <- as.list(extras)
    plot(rx, ry, type="n", axes = FALSE, xlab = "", ylab = "",
         main = main)
    abline(v = ix, h = iy, col = "lightgray", lty = "dotted")
    for(i in 1:np) {
      pc <- pch[[i]]
      ## 'col' symbols with a 'bg'-colored interior (where available) :
      points(ix[i], iy[i], pch = pc, col = col, bg = bg, cex = cex)
      if(cextext > 0)
          text(ix[i] - 0.3, iy[i], pc, col = coltext, cex = cextext)
    }
  }
pchShow()
```

You might also want to add lines to a plot, which can be done in two ways. If you have some data points that you want to draw lines between, you can use `lines`. The first and second arguments are the x and y coordinates of the data and the third, `lty`, indicates the type of line that I want to draw (1 = solid, 2= dashed, 3=dotted, etc.). Note that you have to rerun the plot, since you just overwrote it with `pchshow`.

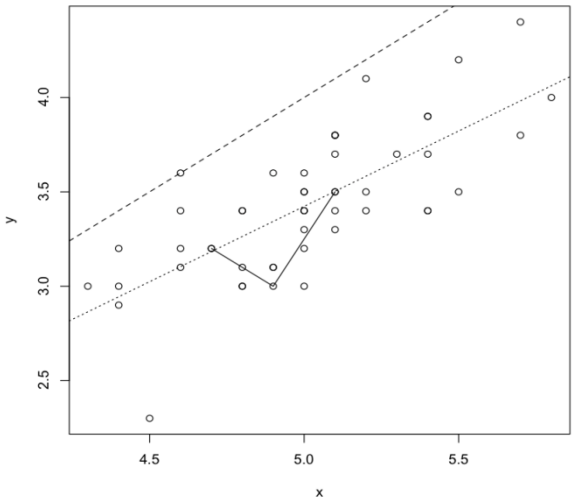
```
plot(x,y)
lines(iris$Sepal.Length[1:3], iris$Sepal.Width[1:3], lty=1)
```

If you want to add a line when you know the intercept and slope, use `abline`, which takes these values as the first and second arguments.

```
abline(-1,1,lty=2)
```

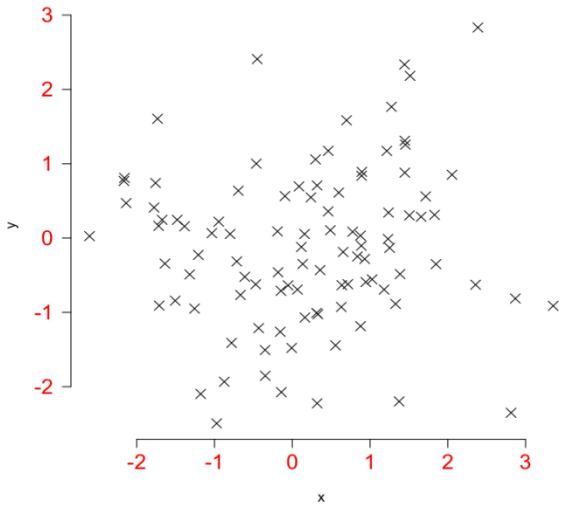
`abline` can also conveniently extract the slope and intercept from regression models. Without worrying about the specification of the regression part, which I discuss in section 3.1, check that the following adds a regression line.

```
abline(lm(iris$Sepal.Width[iris$Species=='setosa']~
iris$Sepal.Length[iris$Species=='setosa']),lty=3)
```



If you've looked in the help file for plot (`?plot`), you've probably noticed that there aren't many arguments listed. This is because they're mostly located in `par`, which is used to set graphical parameters. It's not quite an obvious setup; most arguments for `par` can be fed directly to `plot` (and many other plotting functions). These arguments allow you change the location, sizes, and symbols of just about everything on a figure. Here's an example; see `?par` for more options:

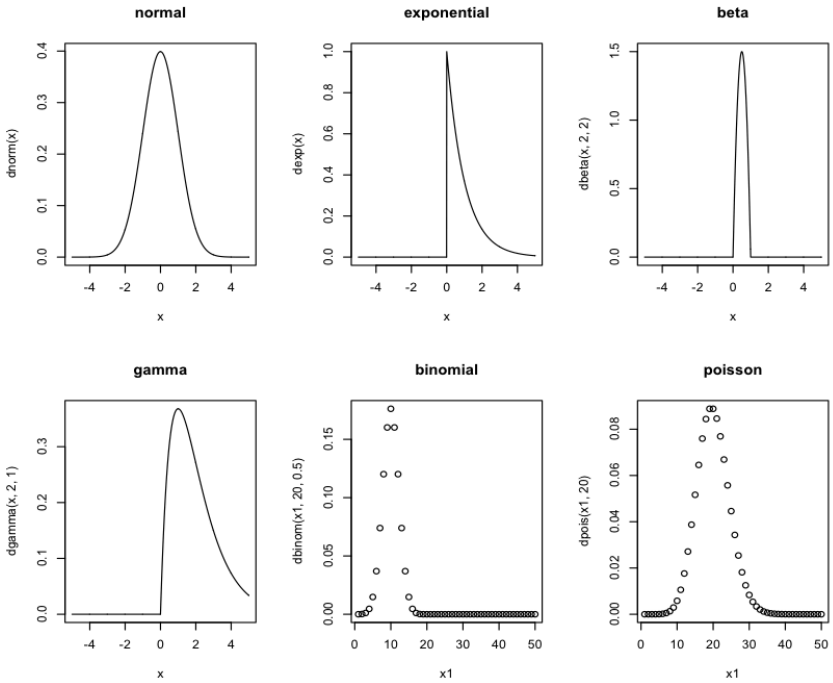
```
x=rnorm(100)
y=rnorm(100)
plot(x, y, bty='n', bg='yellow', cex=1.5, cex.axis=1.5, col.axis='red',
las=1, pch=4)
```



These options can be read as follows: x coords, y coords, no bounding box around figure, background color, magnification of plot symbols, magnification of axis, axis color, axis labels always horizontal, plot symbol type.

It's often nice to put multiple plots on the same figure. This can be achieved with a call to `par` using `mfrow`. For `mfrow`, you must specify the number of rows and columns of plots. The rows and columns will fill with each plotting function you call. Note that once you set an attribute like `mfrow`, your plot window will be stuck with it until you either set it to another value or close the window. The latter is usually easiest, as the next call to a plotting function will return to the default window. Here's an example of `mfrow` to plot the probability density functions of a few common distributions:

```
par(mfrow=c(2,3))
x=seq(-5,5,by=.01) ## for continuous distributions
plot(x,dnorm(x),type='l',main='normal')
plot(x,dexp(x),type='l',main='exponential')
plot(x,dbeta(x,2,2),type='l',main='beta')
plot(x,dgamma(x,2,1),type='l',main='gamma')
x1=1:50 ## for discrete distributions
plot(x1,dbinom(x1,20,.5),main='binomial')
plot(x1,dpois(x1,20),main='poisson')
```



Finally, you can annotate plots using `text` by specifying the x and y coordinates following by the string to plot.

```
par(mfrow=c(1,1))
plot(x1,dpois(x1,20),main='poisson')
text(20,.06,'mean=20')
```

2.2 Histograms

Histograms are bar charts that show the frequency or probability of observing values of your independent variable. We'll construct some histograms using the BCI data set; this is part of the famous data set on tree occurrence from Barro Colorado Island that's been used to both support and rebuke (and everything in between) the Neutral Theory of Biodiversity. This is included in the vegan package, so we load that and `attach` the data. Attaching the data allows you to more succinctly reference the columns of a data set rather than using the `datasetname$` prefix. However, you should use caution when using `attach`, because it can put variables in your workspace that you're unaware of, can overwrite other objects with the same name, and requires that you remember to use attach again when running the code again (so I actually never use it).

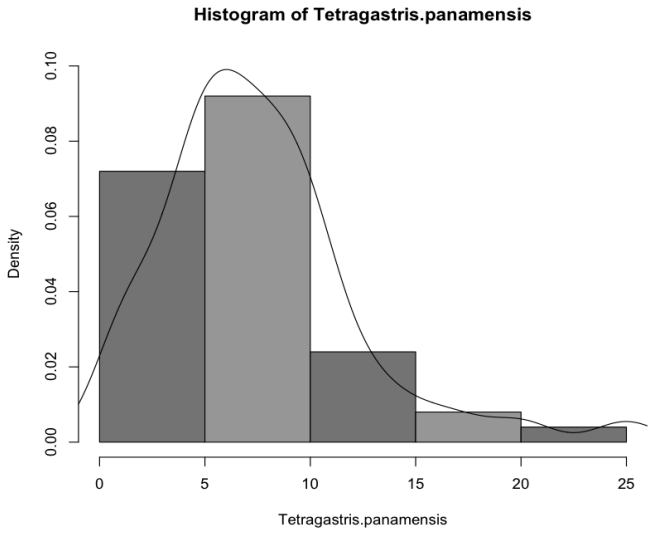
```
library(vegan)
data(BCI)
BCI$Tetragastris.panamensis ## a column of BCI
attach(BCI)
Tetragastris.panamensis ## the same column, referenced after attach
names(BCI) ## the column (species) names of BCI
```

Now I'll demonstrate some of the options of `hist`.

```
par(mfrow=c(2,2))
hist(Tetragastris.panamensis)
hist(Tetragastris.panamensis, freq=F)
hist(Tetragastris.panamensis,breaks=3)
hist(Tetragastris.panamensis,breaks=20)
```

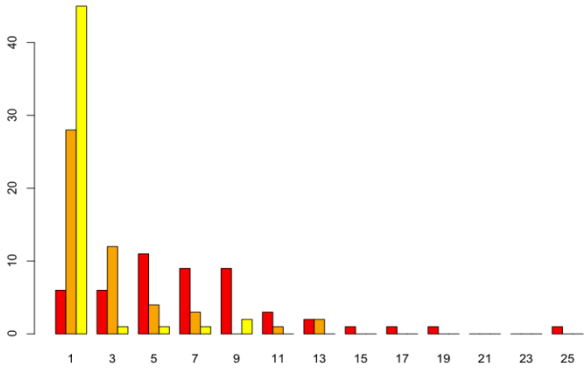
Now we'll add a few bells and whistles to a basic plot. `density` computes a normalized and smoothed empirical density function.

```
par(mfrow=c(1,1))
colors=rep(c('grey45','grey60'),5)
hist(Tetragastris.panamensis, freq=F, col=colors,ylim=c(0,.1))
lines(density(Tetragastris.panamensis))
```



A histogram to compare multiple species can be made using `multhist` from the plotrix package. Note that the data must enter as a list.

```
library(plotrix)
dat=list(a=Tetragastris.panamensis, b=Zanthoxylum.ekmanii, c=
Terminalia.oblonga)
multhist(dat,col=c('red','orange','yellow'))
```



2.3 Box plots

A boxplot provides a graphical view of the median, quartiles, maximum, and minimum of a data set (for more: http://en.wikipedia.org/wiki/Box_plot). Let's generate some data and see what it looks like.

```
w1=rnorm(100)
boxplot(w1)
```

This is a very plain graph, and the title, labels, etc., can be specified in exactly the same way as in the `plot` and `hist` commands:

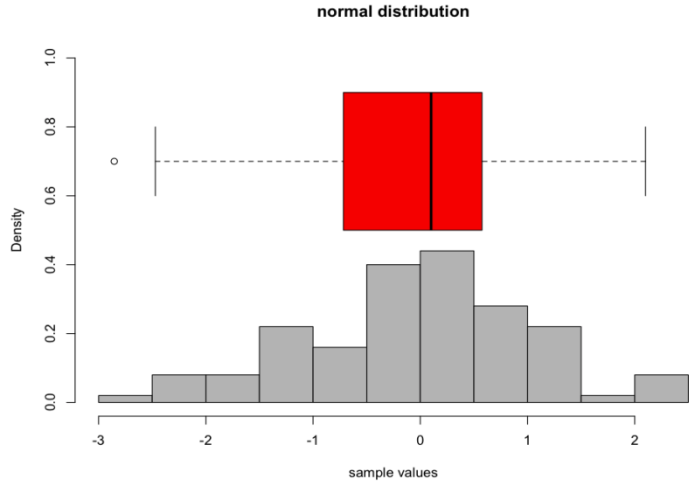
```
boxplot(w1, main='normal distribution', ylab='sample value', col='red')
```

Note that the default orientation is to plot the boxplot vertically. Because of this we used the `ylab` option to specify the axis label. You can specify that the boxplot be plotted horizontally by specifying the horizontal option:

```
boxplot(w1,main='normal distribution', xlab='sample values',
horizontal=TRUE)
```

The option to plot the box plot horizontally can be put to good use to display a box plot on the same image as a histogram. You need to specify the `add` option, specify where to put the box plot using `at`, and turn off the addition of axes using `axes`:

```
hist(w1,main='normal distribution', xlab='sample
values',ylim=c(0,1),freq=F,col='grey70')
boxplot(w1,horizontal=TRUE,at=.7,add=TRUE,axes=FALSE,col='red')
```

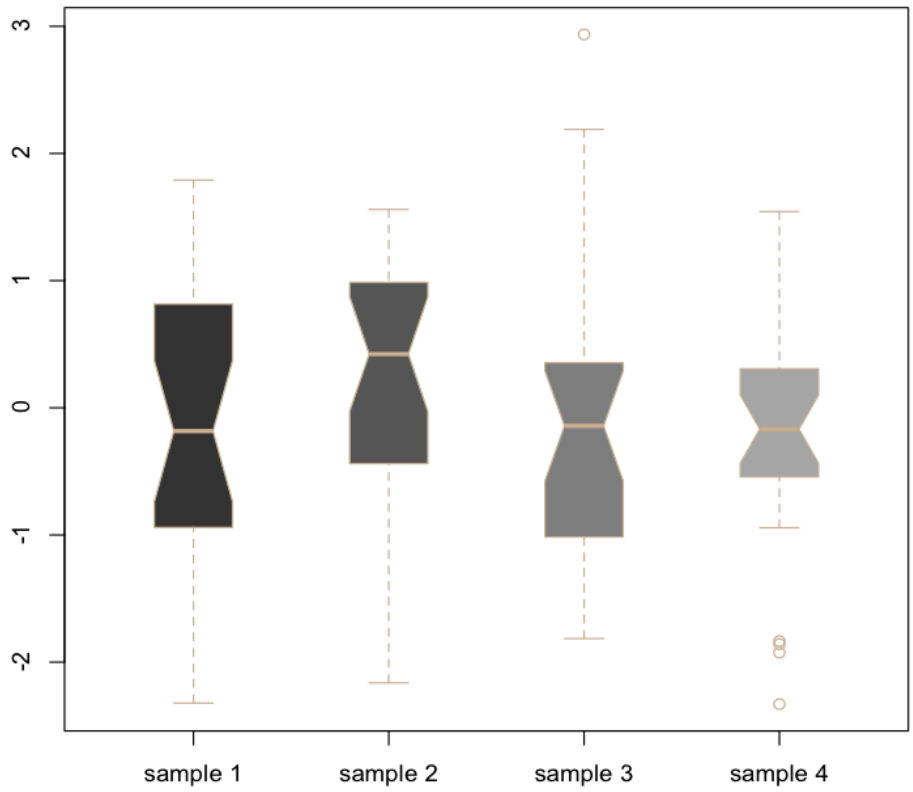


We can also produce multiple box plots to compare data. Say we divide the sample we took from the normal distribution into four equal sized samples, and want to check how similar they are. We can construct a factor that distinguishes which group each sample belongs to, then plot each of the groups.

```
group=factor(rep(1:4,25))
boxplot(w1~group)
```

We can add a few more features to the plot such as notches (if two notches don't overlap, the medians of the groups are probably different), and some decorations. Note that `paste` is useful to combine vectors efficiently. It combines the first argument (here, a vector of length 1 with the character 'sample') with the second argument (the sequence 1,2, 3, 4) and puts the value of `sep` (in this case space) between them. `boxwex` controls the width of the box (.4 decreases them to 40% of default values) and `border` defines the border color for the boxes. Not only is there a color called 'peachpuff', there are multiple variations of 'peachpuff'. Type `colors()` for 656 other options.

```
boxplot(w1~group,
     names=paste('sample',1:4,sep=' '),
     notch=T,col=grey(seq(.2,.8,.15)), boxwex=.4, border='peachpuff3')
```



Finally, I end with a cool example from the R graph gallery (http://addictedtor.free.fr/graphiques/thumbs.php)that shows various ways of representing boxplot data to give you an idea of the possibilities.

```
#generate some data
x = c(rnorm(100,0,1), rnorm(100,3,1))
xxx = seq(min(x), max(x), length=500)
yyy = dnorm(xxx)/2 + dnorm(xxx, mean=3)/2
#plot the data in multiple ways
par(mfrow=c(1,5), mar=c(3,2,4,1))
plot(yyy, xxx, type="l",main="Underlying\ndensity")
boxplot(x, col="gray90", main="standard\nboxplot")
hdr.boxplot(x, main="HDR\nboxplot")
vioplot(x)
title("violin plot")
bpplot(x)
```

Note that if you get an error like

```
Error: could not find function "hdr.boxplot"
```

It's because you haven't loaded the package (`hdrcde`) that contains this function. You can find all the packages you need in section 0.3.

2.4 Three dimensional plots

Representing three dimensional data can be useful to show more complex associations between variables. This can be done using a variety of functions. For example, topographic data from a Digital Elevation Model (DEM) contains x and y coordinates but also elevation. One might show this on a contour plot.

```
x = 10*(1:nrow(volcano))
y = 10*(1:ncol(volcano))
contour(x, y, volcano, levels = seq(90, 200, by = 5), col = "peru")
```

It's worthwhile to have a look at the data structure for this type of plot by typing `volcano`. As you start using more and more add on packages, you'll get in the habit of figuring out the data format required by the package functions. In this case, the z variable, elevation is stored in a matrix where the rows correspond to x coordinates and the columns correspond to y coordinates.

Anyhow, rather than using contours, we could alternatively use different colors to indicate different elevations.

```
image(x, y, volcano, col = terrain.colors(100), axes = FALSE)
```

or use contours and colors together

```
image(x, y, volcano, col = terrain.colors(100), axes = FALSE)
contour(x, y, volcano, add=T, levels = seq(90, 200, by = 5), col =
"peru")
```

Or, we might plot in three dimensions.

```
z = 2 * volcano        # Exaggerate the relief
x = 10 * (1:nrow(z))   # 10 meter spacing (S to N)
y = 10 * (1:ncol(z))   # 10 meter spacing (E to W)
persp(x, y, z, theta = 135, phi = 30, col = "green3", scale = FALSE,
      ltheta = -120, shade = 0.75, border = NA, box = FALSE)
```

3. Regression

3.1 Linear regression

We'll start with a simple example of multivariate regression by ordinary least squares. To make it easier to evaluate model output, we'll simulate some data. Simulating data is a useful strategy when building complex models to make sure that you've set up the model correctly because you know the answer that the model should give ahead of time. To simulate data: (1) specify the predictors (which I draw as random samples from a normal distribution), (2) specify the regression coefficients, (3) multiply these together (using matrix multiplication) to produce the response variable. Matrix multiplication is performed using the %*% operator (follow this link if you don't know what matrix multiplication is http://www.math.umn.edu/~nykamp/m2374/readings/matvecmultex/).

```
     ## first, ensure that my random number generator looks like yours
set.seed(1)
     ## design matrix-rows are samples and columns are predictors
X=data.frame(cbind(rnorm(100),rnorm(100),rnorm(100),rnorm(100)))
names(X)=paste('x',1:4,sep='')
     ## true regression coeficients
beta=c(2,-1,3,.05)
     ## create response variable
y=as.matrix(X)%*%beta+rnorm(100)
```

Note that I had to use `as.matrix` to perform the matrix multiplication because X is technically a data frame even though it has numeric entries in the columns. Also, I added random error in the last line (`rnorm(100)`)so that the model wouldn't fit perfectly. Now we're ready to construct a regression model to see how predictors x1, x2, x3 and x4 predict response variable y. The function `lm` separates the response from the predictors using "~" and uses arithmetic operators (i.e. +, -,*,/) in a special format specific to model formulas. For example for two vectors x1 and x2, writing `x1 + x2` normally adds the values of x1 to x2 and produces a new vector the same length as x1 or x2. But in a formula, writing `y~x1+x2` indicates that both predictors should be included in a multivariate regression. More examples will follow as we build more complex models. We begin with a multivariate regression with four predictors:

```
lm(y~x1+x2+x3+x4,data=X)

# Call:
# lm(formula = y ~ x1 + x2 + x3 + x4, data = X)

# Coefficients:
# (Intercept)           x1           x2           x3           x4
#   -0.05698      2.19333     -0.83036      3.03375      0.09268
```

By specifying the value for 'data' one can simply write the column names for X as the predictors. Alternatively, one can leave out the 'data' argument and equivalently write

```
lm(y~x1+x2+x3+x4,data=X)
```
or
```
lm(y~X[,1]+X[,2]+X[,3]+X[,4])
```

If you want to use all the columns of X as predictors, the following shorthand is nice

```
lm(y~.,data=X)
```

Other operators are available to specify model formulas efficiently. For example, `x1:x2` includes the interaction term for x1 and x2. The arithmetic operator for multiplication (*) also has a special meaning in formulas. Normally writing x1*x2 produces a vector the same length as x1 or x2 with the corresponding rows of x1 and x2 multiplied together. But in a formula, writing `x1*x2` includes both main effects and the interaction term and is equivalent to `x1+x2+x1:x2`. The exponential operator also has a special meaning in models. Writing `(x1+x2+x3+x4)^2` includes the main effects and all possible first order interaction terms and is equivalent to `x1+x2+x3+x4+x1:x2+x1:x3+x1:x4+x2:x3+x2:x4+x3:x4`. For example:

```
lm(y~(x1+x2+x3+x4)^2,data=X)
```

```
# Call:
# lm(formula = y ~ (x1 + x2 + x3 + x4)^2, data = X)

# Coefficients:
# (Intercept)           x1           x2           x3           x4
#   -0.02821      2.07919     -0.79111      3.00478      0.04757
#      x1:x2        x1:x3        x1:x4        x2:x3        x2:x4
#   -0.25167     -0.09652     -0.01544     -0.04378     -0.05302
#      x3:x4
#   -0.14702
```

The minus operator (-) is also helpful for removing terms from the model. For example, writing `(x1+x2+x3+x4)^2 - x1:x2` produces the same model as above with only the x1x2 interaction term removed. Finally, if you do want to perform an arithmetic operation on a predictor, like using the square of x1, this can be specified using `I(x1^2)`. In general writing `I(...)` tells the formula to interpret the operators inside the parentheses in the usual arithmetic form. See `?formula` for more details on succinct ways to express models.

You may have noticed that these models all implicitly include intercept terms. The intercept can be specified a priori as follows. Values other than 0 are also acceptable.

```
lm(y~0+x1+x2+x3+x4,data=X)
```

```
# Call:
# lm(formula = y ~ 0 + x1 + x2 + x3 + x4, data = X)

# Coefficients:
#     x1       x2       x3       x4
# 2.1856  -0.8283   3.0327   0.0896
```

When building models, it's often useful to assign the model a name for reference in other functions. For example, the `summary` function provides useful model attributes.

```
model1=lm(y~x1+x2+x3+x4,data=X)
summary(model1)
```

```
# Call:
# lm(formula = y ~ x1 + x2 + x3 + x4, data = X)

# Residuals:
#     Min       1Q   Median       3Q      Max
# -2.55929 -0.73645 -0.09146  0.68187  3.71009
```

```
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) -0.05698    0.11800  -0.483     0.63
# x1           2.19333    0.13095  16.750  < 2e-16 ***
# x2          -0.83036    0.12292  -6.755 1.14e-09 ***
# x3           3.03375    0.11438  26.523  < 2e-16 ***
# x4           0.09268    0.11949   0.776     0.44
# ---
# Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

# Residual standard error: 1.168 on 95 degrees of freedom
# Multiple R-squared: 0.9192, Adjusted R-squared: 0.9158
# F-statistic: 270.1 on 4 and 95 DF,  p-value: < 2.2e-16
```

In the output above, the first thing we see is the call, this is R reminding us what model we ran, what options we specified, etc.. Next we see the residuals, which are a measure of model fit. The next part of the output shows the coefficients, their standard errors, the t-statistic and the associated p-values. x4 and the intercept are not significant. As an aside, it's worth noting that `summary` can summarize lots of different types of R objects, so you might try it when you want to know more about an object.

The object `model1` has a number of attributes which can be accessed with the $ operator. For example:

```
model1$coefficients
#(Intercept)          x1          x2          x3          x4
#-0.05697542  2.19332866 -0.83036007  3.03374761  0.09268219

model1$fitted.values
#     1          2          3          4          5          6
# 0.40898269  5.33739282  3.86262609  2.27133229 -5.57023367  4.39341304
#      7          8          9         10         11         12
# 2.44839125  2.50146277  0.75128197 -0.54612721  3.38479721  2.46687445 …
```

The fitted values are the predicted values for each observation. For more options type `?lm` and see the Values section.

It is often desirable to use a model to predict the values of the response variables for new values of the predictors (e.g. in cross validation). For example, we might want to project the species distribution model constructed in the next section on to a new region. First, we simulate some new values for the predictors for which we want to predict the response, then we use the predict function by specifying the model to be used and these new predictor values.

```
newdata=data.frame(cbind(rnorm(10),rnorm(10),rnorm(10),rnorm(10)))
names(newdata)=paste('x',1:4,sep='')
predict(model1,newdata)
#     1          2          3          4          5          6
# -0.1352304  1.0592061 -4.5261272  0.5175283  2.4605639 -1.9348440
#          7          8          9         10
# -3.1264920  2.7841574  4.9129509 -3.0829223
```

Note that `predict` requires that predictor (column) names are the same in the original data (X) and the new data (newdata).

Once you've fit one model, it's often useful to compare it to other models to determine which is best. A simple way to do this is with a Chi-squared difference test. The Chi-squared statistic measures the

fit of a model to a set of observations. If this statistic is calculated for two competing models, the model with the lower value is better. One can determine the significance of the difference between the models using a Chi-squared difference test with `anova`. For example, if `model1` contains all (linear) predictors and `model2` omits just x4, the Chi-squared difference test asks whether `model2` is statistically worse (i.e. if the simpler model doesn't lose much prediction accuracy, we might as well keep it). More details on this test are given here http://zencaroline.blogspot.com/2007/05/chi-square-difference-test-for-nested.html (this describes the test for Structural Equation Models (SEM) but the test description doesn't depend on knowing what those are). First, let's determine if omitting x1 from the model might be okay.

```
model1=lm(y~x2+x3+x4,data=X)
model2=lm(y~x1+x2+x3+x4,data=X)
anova(model1,model2,test="Chi")
# Analysis of Variance Table

# Model 1: y ~ x2 + x3 + x4
# Model 2: y ~ x1 + x2 + x3 + x4
#   Res.Df    RSS Df Sum of Sq P(>|Chi|)
# 1     96 512.72
# 2     95 129.69  1    383.02 < 2.2e-16 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The RSS column (Residual Sum of Squares: http://en.wikipedia.org/wiki/Residual_sum_of_squares) indicates how well the model fits by adding up the squared differences between observed and predicted y values. The fourth row (beginning with the number 1) gives the fit of the model without x1. A large RSS means bad fit; model 1 is 512 while model 2 is 129. So it looks like the full model fit better. We can do a significance test to determine whether the improvement in RSS might have occurred by chance (a trivial test in this case). The significance of the comparison indicates that the more complex model is statistically better as determined by the column P(>|Chi|). To summarize, the table says that the probability of observing a decrease in the RSS of 383.02 purely by chance when adding 1 more parameter would occur (2.2 e-16) x 100% = .000000000000022 % of the time. So including x1 is a good idea.

But can we improve on model2 by adding an interaction term between x1 and x4?

```
model3=lm(y~x1+x2+x3+x4*x1,data=X)
anova(model2,model3,test="Chi")
# Analysis of Variance Table

# Model 1: y ~ x1 + x2 + x3 + x4
# Model 2: y ~ x1 + x2 + x3 + x4 * x1
#   Res.Df    RSS Df Sum of Sq P(>|Chi|)
# 1     95 129.69
# 2     94 129.64  1  0.054524    0.8424
```

The RSS isn't much improved in the more complex model (i.e. 129.69 isn't very different from 129.64) so it's unlikely to come up significant. Checking the last column, we see that this comparison is not significant, indicating that the simpler model (`model2`) is better. Such tests are important because increasing the number of predictors can always increase model fit metrics like RSS and R^2, but if this increase is insignificant compared to what we might expect by chance, the simpler model is preferred.

There are a number of functions that quickly allow you to explore different models. The functions

add1 and drop1 consider models that add or drop certain predictors. We might ask whether model2 could be improved upon by dropping a predictor

```
drop1(model2,test='Chi')
# Single term deletions

# Model:
# y ~ x1 + x2 + x3 + x4
#       Df Sum of Sq     RSS     AIC   Pr(Chi)
# <none                 129.69  36.001
# x1      1    383.02  512.72 171.455 < 2.2e-16 ***
# x2      1     62.30  191.99  73.229 3.770e-10 ***
# x3      1    960.34 1090.04 246.880 < 2.2e-16 ***
# x4      1      0.82  130.52  34.632    0.4269
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The insignificance of x4 indicates that a model omitting x4 is preferred.

If you have a very large suite of potential predictors, stepwise regression is often useful to determine a simple model. Forward stepwise regression considers whether adding a single predictor to more predictors produces a significantly better model. If we begin with `model2` and wanted to consider the value of 4 additional predictors, forward stepwise regression adds each predictor, one at a time, to model2 and asks whether that predictor provides significant improvement over model2. The first 'step' consists of comparing model2 to each of the 4 other possible models that contain one additional predictor. From these, the best model is retained. The second step starts with the best model from the previous step and compares it to the 3 possible models that contain one additional predictor. The steps proceed until none of the proposed models are better than the best model from the previous step.

Backward stepwise regression performs the same procedure but instead begins with a complex model and sequentially removes each predictor and asks whether the simpler model is better. It is also possible to build a model with both forward and backward steps simultaneously. It is important to note that a limitation of stepwise regression is that it can only consider models that add or drop one predictor and that this search procedure is not guaranteed to produce the best possible model from the suite of all possible predictors. Stepwise regression is therefore usually used for data exploration or when considering all possible models constructed from a large set of predictors is impractical.

Stepwise regression is typically performed using AIC (http://en.wikipedia.org/wiki/Akaike_information_criterion) as the model fit metric because it penalizes models for having more parameters. To implement a stepwise regression, we can simulate a new suite of predictors and then use the function `step`. For data exploration, it's typically a good idea to start with the most complex model and run both forward and backward selection simultaneously.

```
X2=data.frame(cbind(rnorm(100),rnorm(100),rnorm(100),rnorm(100)))
names(X2)=paste('x',5:8,sep='')
model5=lm(y~.,data=cbind(X,X2))
step(model5,~.,direction='both',data=X2)
#… I omit the lengthy output
#…
# Step:  AIC=34.63
```

```
# y ~ x1 + x2 + x3

#       Df Sum of Sq     RSS     AIC
# <none                  130.52  34.632
# + x4    1      0.82  129.69  36.001
# + x6    1      0.55  129.96  36.207
# + x7    1      0.28  130.23  36.416
# + x5    1      0.15  130.37  36.517
# + x8    1      0.00  130.51  36.631
# - x2    1     63.25  193.76  72.146
# - x1    1    382.22  512.73 169.459
# - x3    1    978.18 1108.69 246.576

# Call:
# lm(formula = y ~ x1 + x2 + x3, data = cbind(X, X2))

# Coefficients:
# (Intercept)           x1           x2           x3
#   -0.05209      2.18796     -0.83545      3.04344
```

Above I show only the last step, which finds that the best model includes only x1, x2 and x3.

Finally, we can check a few model diagnostics easily. If we take our best model, which includes only x1, x2, x3, we can use the plot function to check model assumptions.

```
model6=lm(y~x1+x2+x3,data=X)
plot(model6)
```

Follow the prompts on the command line to scroll through the output. The first plot determines where the residual variance is constant over the range of the response. The second determines whether the normality assumption of residuals is reasonable. The third plot gives another measure of residual variance and the fourth identifies high leverage points that dominate model fitting.

3.2 Generalized linear regression

Generalized linear models (GLMs) make linear regressions more general by allowing two additional features: (1) response variables that are not continuous or normally distributed, and (2) a nonlinear relationship between predictors and response represented by a 'link' function. For example, binary data consists of 0's and 1's and might represent presence and absence of a distribution. This requires that we assume a binomial distribution (http://en.wikipedia.org/wiki/Binomial_distribution) for the response variable. Count data consists of positive integers and should be modeled with a Poisson distribution (http://en.wikipedia.org/wiki/Poisson_distribution).

Nonlinear relationships between predictors and response are represented using different link functions. For example, an 's' shaped function might better represent the relationship between the response and predictors than a linear function when the expectation value for the response represents a probability of survival. Link functions are also useful to restrict the expectation value of the response to lie on some meaningful interval, as when a survival probability is constrained to lie on [0,1]. Most distributions for the response variable are commonly used with a particular link function: a logit link is used with binomial or multinomial data, a log link is used with count (Poisson) data, and an inverse link (1/y) is used with exponential data. A model with a logit link, coefficients *B*, and error *e* is usually written symbolically in the literature as:

$$\text{logit}(y) \sim XB + e$$

A technical description of GLMs is here: http://en.wikipedia.org/wiki/Generalized_linear_model, and a less formal one is here:

http://userwww.sfsu.edu/~efc/classes/biol710/Glz/Generalized%20Linear%20Models.htm. The logit function is the logarithm of the odds ratio between presence and absence. The odds ratio is just the probability of presence, *p*, divided by the probability of absence, *1-p*. Larger values of this ratio mean that presence is more likely. Taking the logarithm of this odds ratio doesn't change this relationship.

As an example, we'll construct a species distribution model. The observations represent presences (1's) and absences (0's). Environmental predictors include the mean annual precipitation (map) the minimum annual temperature (min.t) and an index of soil fertility (soil) at each site. The model seeks the probability of presence of the species as a function of the environmental covariates. This corresponds to a binomial distribution for the response data, where we model the mean of the binomial distribution (*p*) as a function of the environmental covariates. The 1's and 0's that we have as observations therefore represent a random sample of from a binomial distribution with mean *p*, which we could write as *p(map, min.t, soil)* to indicate the dependence on covariates.

First we download and read in the data (you must be connected to the internet). I've adapted this example from UCLA's Academic Technology Services R resources - check out their well-documented resources for more examples (http://www.ats.ucla.edu/stat/R/).

```
mydata = read.csv(url("http://www.ats.ucla.edu/stat/r/dae/binary.csv"))
names(mydata)=c('pres','map','min.t','soil')
attach(mydata)
```

This dataset has a binary response (presence, absence) variable called `pres`. There are three predictor variables: `map, min.t` and `soil`. `map` and `min.t` are continuous variables, while `soil` is categorical and takes on the values 1 through 4, indicating fertility. Sites with a value of 1 have the highest soil fertility, while those with a soil of 4 have the lowest. First we look at a few summaries of the data

```
summary(map)
#  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#  220.0   520.0   580.0   587.7   660.0   800.0

summary(min.t)
#  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#  2.260   3.130   3.395   3.390   3.670   4.000

table(soil)
#soil
#  1   2   3   4
# 61 151 121  67

table(pres)
#pres
#  0   1
#273 127

table(soil,pres)
#    pres
#soil  0  1
#   1 28 33
#   2 97 54
#   3 93 28
#   4 55 12
```

The code below estimates a logistic regression model using the glm (generalized linear model) function. The as.factor(soil) indicates that soil should be treated as a categorical variable. If there are N categories, this means that a coefficient is estimated for N-1 of the categories, hence you'll see multiple predictors related to soil below. A brief description of categorical variables in regression can be found here: http://www.ats.ucla.edu/stat/r/modules/dummy_vars.htm .

```
mylogit= glm(pres~map+min.t+as.factor(soil),
family=binomial(link="logit"))
```

Since we gave our model a name (mylogit), R will not produce any output from our regression. In order to get the results we use the summary command:

```
summary(mylogit)
```

```
# glm(formula = pres ~ map + min.t + as.factor(soil), family = binomial(link = "logit"),
#     na.action = na.pass)
#
# Deviance Residuals:
#     Min       1Q   Median       3Q      Max
# -1.6268  -0.8662  -0.6388   1.1490   2.0790
#
# Coefficients:
#                   Estimate Std. Error z value Pr(>|z|)
# (Intercept)      -3.989979   1.139951  -3.500 0.000465 ***
# map               0.002264   0.001094   2.070 0.038465 *
# min.t             0.804038   0.331819   2.423 0.015388 *
# as.factor(soil)2 -0.675443   0.316490  -2.134 0.032829 *
# as.factor(soil)3 -1.340204   0.345306  -3.881 0.000104 ***
# as.factor(soil)4 -1.551464   0.417832  -3.713 0.000205 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# (Dispersion parameter for binomial family taken to be 1)
#
#     Null deviance: 499.98  on 399  degreesof freedom
# Residual deviance: 458.52  on 394  degreesof freedom
# AIC: 470.52

# Number of Fisher Scoring iterations: 4
```

In the output above, the first thing we see is the call, this is R reminding us what model we ran, what options we specified, etc.. Next we see the deviance residuals, which are a measure of model fit. This part of the output shows the distribution of the deviance residuals for individual cases used in the model. The next part of the output shows the coefficients, their standard errors, the z-statistic (sometimes called a Wald z-statistic), and the associated p-values. Both map and min.t are statistically significant, as are the three terms for soil.

The logistic regression coefficients give the change in the log odds of the outcome for a one unit increase in the predictor variable. Recall that log odds is just another name for the logit function, discussed above. For every one unit change in map, the log odds of presence (versus absence) increases by 0.002. For a one unit increase in min.t, the log odds of presence increase by 0.804. The indicator variables for soil have a slightly different interpretation. For example, soil fertility 2, compared to fertility 1, decreases the log odds of admission by 0.675. Below the table of coefficients are fit indices, including the null and deviance residuals and the AIC. Later we show an example of how you can use these values to help assess model fit.

We can use confit to obtain confidence intervals for the coefficient estimates.

```
confint(mylogit)
# Waiting for profiling to be done...
#                             2.5 %       97.5 %
# (Intercept)      -6.2716202334 -1.792547080
# map               0.0001375921  0.004435874
# min.t               0.1602959439  1.464142727
# as.factor(soil)2 -1.3008888002 -0.056745722
# as.factor(soil)3 -2.0276713127 -0.670372346
# as.factor(soil)4 -2.4000265384 -0.753542605
```

You can also exponentiate the coefficients and interpret them as odds-ratios.

```
exp(mylogit$coefficients)
```

```
#     (Intercept)              map            min.t as.factor(soil)2
#       0.0185001        1.0022670        2.2345448        0.5089310
# as.factor(soil)3 as.factor(soil)4
#       0.2617923        0.2119375
```

Now we can say that for a one unit increase in min.t, the odds of being present (versus absent) increase by a factor of 2.23. For more information on interpreting odds ratios see http://www.ats.ucla.edu/stat/mult_pkg/faq/general/odds_ratio.htm . The confidence intervals around the odds ratios can be obtained using the `exp(confint(mylogit))`.

You can also obtain the predicted probability of presence for different values of your predictors to help understand the model. In order to create predicted probabilities we first need to create a new data frame with the values we want the predictors to take. We will start by calculating the predicted probability of admission at each value of soil, holding map and min.t at their means. These objects must have the same names as the variables in your logistic regression above (e.g. in this example the mean for map must be named map).

```
newdata1 = data.frame(map = mean(mydata$map), min.t =
mean(mydata$min.t), soil = c(1,2,3,4))
newdata1 #show the new data frame
```

```
#      map  min.t   soil
# 1 587.7 3.3899    1
# 2 587.7 3.3899    2
# 3 587.7 3.3899    3
# 4 587.7 3.3899    4
```

Next we tell R to create the predicted probabilities. Below, the newdata1$soilP creates a new variable in the data frame newdata1 called soilP, the rest of the command tells R that the values of soilP should be predictions made using `predict`. The arguments of `predict` indicate that predictions are based on the analysis mylogit with values of the predictor variables coming from newdata1 and that the type of prediction is a predicted probability (type="response").

```
newdata1$soilP =predict(mylogit,newdata=newdata1,type="response")
newdata1
```

```
#      map  min.t soil soilP
# 1 587.7 3.3899    1 0.5166016
# 2 587.7 3.3899    2 0.3522846
# 3 587.7 3.3899    3 0.2186120
# 4 587.7 3.3899    4 0.1846684
```

In the above output we see that the predicted probability of presence is 0.52 in the best soil (soil=1), and 0.18 in the poorest soil (soil=4), holding map and min.t at their means.

We can do something very similar to create a table of predicted probabilities varying the value of map. This time we vary map while holding min.t at its mean and soil fertility at 2. Once again, note that you can choose any values you like for these predictors.

```
newdata2=data.frame(map=seq(200,800,100),min.t=mean(mydata$min.t),soil=
2)
```

The code to generate the predicted probabilities (the first line below) is the same as before, except that the names of the variable and data frame have been changed. Reading the table below we can see that the predicted probability of getting accepted is only .18 if map is 200 and increases to .47 if map is 800 (while min.t is held constant at its mean and soil is held constant at 2).

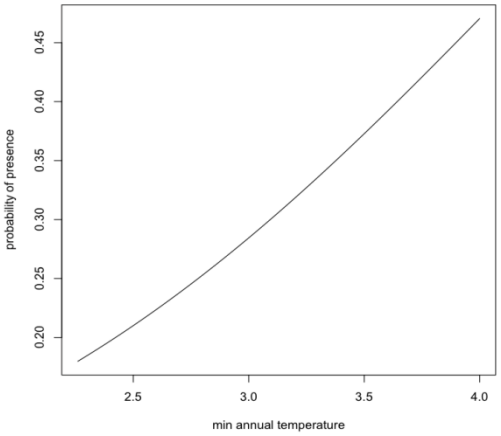
```
newdata2$mapP=predict(mylogit,newdata=newdata2,type="response")
newdata2
```

```
#   map  min.t soil      mapP
# 1 200 3.3899    2 0.1843830
# 2 300 3.3899    2 0.2208900
# 3 400 3.3899    2 0.2623007
# 4 500 3.3899    2 0.3084017
# 5 600 3.3899    2 0.3586658
# 6 700 3.3899    2 0.4122390
# 7 800 3.3899    2 0.4679753
```

It can also be helpful to use graphs of predicted probabilities to understand and/or present the model. Let's construct a new data set that allows us to see the change in predicted probabilities across the range of min.t values.

```
newdata3=data.frame(map=mean(mydata$map),min.t=seq(min(mydata$min.t),ma
x(mydata$min.t),length=100),soil=2)
newdata3$min.tP=predict(mylogit,newdata=newdata3,type="response")
plot(newdata3$min.t,newdata3$min.tP,type='l',ylab='probability of
presence',xlab='min annual temperature')
```

We can also compare two models based on their AIC values. AIC rewards model for having good fit (based on the likelihood) and penalizes them for complexity (number of parameters) to avoid overfitting. (more info here: http://en.wikipedia.org/wiki/Akaike_information_criterion). Lower AIC values indicate better models, and differences greater than 10 usually indicate that there is no support for the model with higher AIC. To check our model against simpler alternatives, we'll construct a simpler model that omits the soil predictor.




```
mylogit2= glm(pres~map+min.t, family=binomial(link="logit"))
AIC(mylogit)
AIC(mylogit2)
```

It appears that it's best to keep soil predictors in the model.

3.3. Summary of standard built-in models in R

Below is a summary of the basic modeling functions in R taken directly from Crawley (2008). Many other more specific models are available in user contributed packages.

lm fits a linear model with normal errors and constant variance; generally this is used for regression analysis using continuous explanatory variables.

aov fits analysis of variance with normal errors, constant variance and the identity link; generally used for categorical explanatory variables or ANCOVA with a mix of categorical and continuous explanatory variables.

glm fits generalized linear models to data using categorical or continuous explanatory variables, by specifying one of a family of error structures (e.g. Poisson for count data or binomial for proportion data) and a particular link function.

gam fits generalized additive models to data with one of a family of error structures (e.g. Poisson for count data or binomial for proportion data) in which the continuous explanatory variables can (optionally) be fitted as arbitrary smoothed functions using non-parametric smoothers rather than specific parametric functions.

lme and lmer fit linear mixed-effects models with specified mixtures of fixed effects and random effects and allow for the specification of correlation structure amongst the explanatory variables and autocorrelation of the response variable (e.g. time series effects with repeated measures). lmer allows for non-normal errors and non-constant variance with the same error families as a GLM.

nls fits a non-linear regression model via least squares, estimating the parameters of a specified non-linear function.

nlme fits a specified non-linear function in a mixed-effects model where the parameters of the non-linear function are assumed to be random effects; allows for the specification of correlation structure amongst the explanatory variables and autocorrelation of the response variable (e.g. time series effects with repeated measures).

loess fits a local regression model with one or more continuous explanatory variables using non-parametric techniques to produce a smoothed model surface.

tree fits a regression tree model using binary recursive partitioning whereby the data are successively split along coordinate axes of the explanatory variables so that at any node, the split is chosen that maximally distinguishes the response variable in the left and right branches. With a categorical response variable, the tree is called a classification tree, and the model used for classification assumes that the response variable follows a multinomial distribution.

4. Other things of interest to ecologists

In this section I briefly mention some useful packages and analyses for ecologists.

vegan – Ordination tools for community ecology – Constrained Correspondence Analysis, Detrended Correspondence Analysis, K-means partitioning, Rank-abundance models

ecodist – Dissimilarity metrics for community matrices.

cluster – For every variety of clustering algorithm you can imagine.

BiodiversityR- This package provides a GUI and some utility functions for statistical analysis of biodiversity and ecological communities, including species accumulation curves, diversity indices, Renyi profiles, GLMs for analysis of species abundance and presence-absence, distance matrices, Mantel tests, and cluster, constrained and unconstrained ordination analysis.

SDMtools - This package provides a set of tools for post processing the outcomes of species distribution modeling exercises. It includes novel methods for comparing models and tracking changes in distributions through time. It further includes methods for visualizing outcomes, selecting thresholds, calculating measures of accuracy and landscape fragmentation statistics, etc.

popbio - Popbio is a package for the construction and analysis of matrix population models. First, the package consists of the R translation of Matlab code found in Caswell (2001) or Morris and Doak (2002).

sp – Constructs all sorts of objects useful for spatially explicit analyses.

FD- FD is a package to compute different multidimensional functional diversity (FD) indices. It implements a distance-based framework to measure FD that allows any number and type of functional traits, and can also consider species relative abundances. It also contains other tools for functional ecologists (e.g. maxent).

asbio - Contains functions from the book: ``Applied Statistics for Biologists''

eco - eco implements the Bayesian and likelihood methods proposed in Imai, Lu, and Strauss (2008b) for ecological inference in 2 × 2 tables as well as the method of bounds introduced by (Duncan and Davis, 1953). The package fits both parametric and nonparametric models using either the Expectation-Maximization algorithms (for likelihood models) or the Markov chain Monte Carlo algorithms (for Bayesian models).

ecolMod - Figures, data sets and examples from the book “A practical guide to ecological modeling - using R as a simulation platform” by Karline Soetaert and Peter MJ Herman (2009). Springer. All figures from chapter x can be generated by “demo(chapx)”, where x = 1 to 11. The R-scripts of the model examples discussed in the book are in subdirectory “examples”, ordered per chapter. Solutions to model projects are in the same subdirectories.

simecol - simecol is an object oriented framework to simulate ecological (and other) dynamic systems. It can be used for differential equations, individual-based (or agent-based) and other models as well.

BIOMOD – for Running various species distribution models and constructing ensembles of models. Note that sometimes it doesn't show up in the package manager, but to install you can use:
`install.packages("BIOMOD",repos="http://R-Forge.R-project.org",dependencies=T)`

demogR – a package for analyzing age-structured population models in R. The package includes tools for the construction and analysis of matrix population models. In addition to the standard analyses commonly used in evolutionary demography and conservation biology, demogR contains a variety of tools from classical demography. This includes the construction of period life tables, and the generation of model mortality and fertility schedules for human populations. The tools in demogR are generally applicable to age-structured populations but are particularly useful for analyzing problems in human ecology.

SDMtools - This package provides a set of tools for post processing the outcomes of species distribution modeling exercises. It includes novel methods for comparing models and tracking changes in distributions through time. It further includes methods for visualizing outcomes, selecting thresholds, calculating measures of accuracy and landscape fragmentation statistics, etc.

dismo - This package implements a few species distribution models, including an R link to the 'maxent' model, and native implementations of Bioclim and Domain. It also provides a number of functions that can assist in using Boosted Regression Trees.

spBayes - For spatially referenced data collected over a fixed set of locations with coordinates in a region of study. Such point–referenced or geostatistical data are often best analyzed with Bayesian hierarchical models. Unfortunately, fitting such models involves computationally intensive Markov chain Monte Carlo (MCMC) methods whose efficiency depends upon the specific problem at hand. This requires extensive coding on the part of the user and the situation is not helped by the lack of available software for such algorithms. spBayes implements a generalized template encompassing a wide variety of Gaussian spatial process models for univariate as well as multivariate point–referenced data.

raster - The raster package provides classes and functions to manipulate geographic (spatial) data in 'raster' format. Raster data divides space into cells (rectangles; pixels) of equal size (in units of the coordinate reference system). Such data are also referred to as 'grid' data. The package should be particularly useful when using very large datasets that cannot be loaded into the computer's memory. Functions will work correctly, because they process large files in chunks, i.e., they read, compute, and write blocks of data, without loading all values into memory at once.

coda - Output analysis and diagnostics for Markov Chain Monte Carlo simulations.