A SHORT R TUTORIAL

Steven M. Holland

Department of Geology, University of Georgia, Athens, GA 30602-2501

May 2008

Installing R

The home page for R is http://www.r-project.org and you can obtain compiled versions of R (called precompiled binary distributions or binaries) for OS X, Linux, and Windows by following the link to CRAN, the Contributed R Archive Network, and selecting a nearby mirror. You can also opt to download the source code and compile R yourself, but this is not recommended for most users. Each link for OS X, Linux, and Windows has a FAQ page specific to that operating system, as well as a more general R FAQ that are both worth reading. In addition, the Manuals link on the left side of the R home page leads to a series of resources (more on these below). In particular, the link for "R Administration and Installation" provides detailed instructions for installation on all operating systems.

OS X users should follow the OS X link and then select the file corresponding to the most recent version of R. This will download a diskimage with a file called R.mpkg, which you should double-click to install R. R can be run from the R.app program, from the command line in Terminal.app, or from the command line in X11.

Linux users should follow the links for their distribution and version of Linux and then download the most recent version of R. Typically, there is a readme file for each distribution and version of Linux that explains the download and installation process.

Windows users should follow the link for Windows and then the link for the base package. As of these notes, the most recent version was R-2.2.1-win32.exe. There is a read me file which contains all of the installation instructions.

Learning R

There are an increasing number of books on R. Several that I've had a chance to read are listed below, from the more basic to the more advanced. All of these attempt to teach R and statistics and do so to varying degrees of success. As a result, I've found that I learn some R and some statistics from each, but all also have their weak points. I highly recommend looking through any book on R carefully before buying it.

Statistics : An Introduction using R, by Michael J. Crawley, 2005. John Wiley & Sons, paperback, 342 p. ISBN: 0470022981. Amazon Price: $44.95

Using R for Introductory Statistics, by John Verzani, 2004. Chapman & Hall/CRC, hardcover, 432 p. ISBN: 1584884509. Amazon Price: $44.95

An R and S-Plus® Companion to Multivariate Analysis, by Brian S. Everitt, 2005. Springer, hardcover, 221 p. ISBN: 1852338822. Amazon Price: $69.95

Data Analysis and Graphics Using R, by John Maindonald, 2003. Cambridge University Press, hardcover, 400 p. ISBN: 0521813360. Amazon Price: $60.17

In addition, there are a very large number of manuals and tutorials available for R online and a Google search will reveal probably more than you care to read. The manuals link on the R home page has links to "An Introduction to R", "R Data Import/Export", and "The R Reference Index", all of which can be useful. An Introduction to R is highly recommended as a basic source of information on R. R Data Import/Export is useful for understanding the many ways in which data may be imported into or exported from R. The R Reference Index is a large pdf (nearly 2500 pages) with a comprehensive listing all of the help files. This can also be purchased for a modest cost from amazon.com, but you may find that the help function in R is sufficient for access to this information.

Basics

R stores data in objects. There are several types of objects, the most basic of which is a variable, which holds the value of a single number or label.

There are two assignment operators. Although the equals sign is more intuitive, the arrow (less than, followed by dash) is much more common. The **>** sign at the beginning of each line is the R prompt. Do not type the prompt, but type everything that follows it.

```
> x = 3

> x <- 3
```

To display the value of any variable, simply type that variable at the prompt.

```
> x
```

Arithmetic operators follow standard symbols for addition, subtraction, multiplication, division, exponents

```
> x <- 3 + 4

> x <- 2 - 9

> x <- 12 * 8

> x <- 16 / 4

> x <- 2 ^ 3
```

Comments are always preceded by a # sign; what follows is never executed.

```
# x <- 2 * 5; this entire line is a comment and will
```

```
# not execute

> x <- 2 * 5    # comments can also be placed after
# something that will execute
```

Note that extra spaces are generally ignored, although including them can make code easier to read.

```
> x<-3+7/14

> x <- 3 + 7 / 14
# this produces the same result as the previous line
```

Capitalization generally matters, as R is case-sensitive. This is particularly important when calling functions, discussed below.

```
> x <- 3

> x
# this will correctly display the value of 3

> X
# this will produce an error if X has not been used
# before.  Worse, this will show the value of X if
# X has been used before (when you really wanted the
# value of x).
```

By using the up arrow key, you can access previously typed commands, to save time typing. These commands can also be edited. Use the down arrow to move back towards commands more recently typed, if you overshoot.

Functions

R has a rich set of functions, which are called with any parameters in parentheses. For example the maximum of two numbers would be called as:

```
> max(9, 5)

> max(x, 4)
# objects can be inserted into functions as well
```

In some cases, no parameter is required, but the parentheses are always required. For example, the function **objects()**, which is used to display all the R objects you have created, does not require any parameters and would be called like this:

```
> objects()

> objects
# this would return an error, since the parentheses are
```

```
    # missing
```

Functions usually return a value. If the command is called without an assignment, then the returned value will be displayed on the screen. If an assignment is made, the value will be assigned to that variable.

```
> max(9,5)
# this will display the maximum of 9 and 5 on the screen

> myMax <- max(9,5)
# this will store the maximum of 9 and 5 in myMax, but not
# display the result on the screen
```

Functions can be nested within a single line, where the result of one function is used as a parameter in another function. Nesting functions too deeply can make the command long, hard to interpret, and hard to debug, if something should go wrong.

```
> y <- sd(max(x) * abs(min(x)) * mean(x) * x)
```

To get help with any function, use the `help()` function or the ? operator. The help pages are often loaded with options that may not be apparent, plus background on how a function works, references to the primary literature, and helpful examples that illustrate how a function could be used.

```
> help(max)

>? max
# could also type ?max without the space
```

You can write your own functions. For example, this uses the `sum()` function, which adds the numbers in vector, and the `length()` function, which counts the elements in a vector, to form a function that calculates the average of a set of numbers. Note that what is in parentheses after function is a list of the parameters that the function will need. Also note that all of the contents of the function are enclosed in curly braces. The indentation is simply to make the function easier to read and is not required.

```
> myAverageFunction <- function (x) {
  mySum <- sum(x)
  mySampleSize <- length(x)
  myAverage <- mySum / mySampleSize
  myAverage
}
```

The parameters of a function can be called in two ways, by name and by position. By name uses a series of assignments, such as:

```
> myFunction(parameterOne = x, parameterTwo = y, parameter
  Three = z)
```

By position hides the parameter names and works only if the values are used in the correct order, that is, the order as stated in the definition of the function, which you could see on the help page. For example,

```
> myFunction(x, y, z)
```

will work if x is meant to be assigned to parameterOne, y is meant to be assigned to parameterTwo, and z is meant to be assigned to parameterThree. Calling a function by position is a quick shorthand, but it is prone to errors because the assignments aren't explicit. On the other hand, calling a function by name is useful because the parameters can be assigned in any order. For example, both of the following will work:

```
> myFunction(parameterOne = x, parameterTwo = y, parameter
    Three = z)

> myFunction(parameterTwo = y, parameterOne = x, parameter
    Three = z)
```

Some functions have default values for the parameter and these can be ignored when calling the function, if desired. See the help page for a function to see which parameters have default assignments

There are a couple of useful functions for manipulating objects in R. To show all of the current objects, use **objects()**. To remove objects, use **remove()**.

```
> objects()
# displays the names of all objects currently in use

> ls()
# also displays the names of all the objects

> remove(someObject)
# removes the object named someObject from use - it cannot
# be retrieved

> remove(list=ls())
# removes all objects, essentially returning you to a blank
# slate in R, although the history of commands you executed
# is preserved
```

Entering Data

For small problems, the quickest way to enter data is by assignment

```
> x <- 14

> y <- 8
```

R has a second type of object, called a vector, which you will use frequently. A vector is simply a list of values and may be numeric or text. Data can be entered into a vector easily by using the `c()` function (c as in concatenate). Short vectors are easily entered this way, but long ones are more easily imported (see below).

```
> x <- c(2,7,9,4,15)
```

A third type of R object is a matrix, which has multiple rows and columns. Small matrices can also be entered easily by using the `matrix()` function. Large ones could also be entered this way, but are also more easily imported (see below).

```
> x <- matrix(c(3,1,4,2,5,8), nrow=3)
```

This will generate a matrix of 3 rows and therefore 2 columns, since there are six elements in the matrix. By default, the values are entered by column, such that column 1 is filled first from top to bottom, then column 2, etc. Thus, this matrix would correspond to:

```
3    2
1    5
4    8
```

The values can be forced to be read by rows instead, with row 1 filled first, from left to right, then row 2, etc., by adding byrow=TRUE to the command:

```
> x <- matrix(c(3,1,4,2,5,8), nrow=3, byrow=TRUE)
```

For long vectors, it is usually more convenient to enter data in Excel, check it for errors there, save the file as a text file (tab-delimited or comma-delimited), and then import it into R for analysis. To read a vector into R, use the `scan()` function. Be sure to put the file and path in quotes.

```
> myData <- scan(file="myDataFile.txt")
# will work only if myDataFile.txt is in R's current directory

> myData <-
   scan(file="/Users/myUserName/Documents/myDataFile.txt")
# an absolute path can also always be used
```

For large matrices, it is likewise more convenient to enter the data in Excel and import it into R for analysis. To read a matrix into R and save it as a data frame (more on data frames below), use the read.table function. There are many options for this function (see the help page), but the following would assume that the first row contains the names of the variables (header=TRUE), that sample names are present and stored in the first column (row.names=1), and that each column is separated with a comma (sep=","). Note that this assumes the data has been set up in Excel with columns corresponding to variables and rows corresponding to samples. If the first line is used to hold the names of the variables and sample names are used in the first column, there should be no entry for that particular row and column (i.e., that Excel cell should be empty). Variable names can be long (at least 34 characters) and needn't

be single words; during import, any blank spaces in a variable name will be replaced with a period in R.

```
> myData <- read.table(file="myDataFile.txt", header=TRUE,
    row.names=1, sep=",")
```

R's `edit()` function can be used to bring up a text editor, which can be easily adapted to a spreadsheet-like environment for entering tabular data, by combining it with the `data.frame()` command. In the new window that appears, the plus button is used for adding variables, the minus button for deleting variables, the blue box for adding rows, and the red x for deleting rows. Click the close box when finished to use your entries. Although this can be used to enter new data, you will likely find it cumbersome and find it easier to enter data in a real spreadsheet, save the results as described above and import them into R. The `edit()` function can be used for fixing data errors, but you may also find that you can do this easily in R without calling `edit()`.

```
> x <- edit(x)
# this will allow you to edit x; the assignment to x is
# necessary to save your results

> xnew <- edit(data.frame())
# this can be called to create a new, empty matrix to
# be filled
```

Accessing Data

Elements of a vector can be called by using subscripts. For example, to see the first element of x, you would type:

```
> x <- c(3, 7, -8, 10, 15, -9, 8, 2, -5, 7, 8, 9, -2, -4, -1,
    -9, 10, 12)

> x[1]
# this will display the first element of x, in other words, 3
```

This can be used to select more than one element, including a shorthand notation for seeing a range of elements.

```
> x[c(1,3,5,7)]
# this will let you see the first, third, fifth, and seventh
# elements. Be sure to include the c(), or this will return
# an error because it thinks you are accessing a four-
# dimensional array

> x[3:10]
# this will let you see the 3rd through 10th elements

> x[c(3:7, 5:9)]
# this will display the 3rd through the 7th elements,
```

```
# then the 5th through the 9th
```

You can use conditional logic to select elements meeting only certain criteria

```
> x[x>0]
# will display only the positive elements in x

> x[x<3]
# only values less than 3

> x[x>0 & x<3]
# all values greater than zero and less than 3

> x[x>=2]
# greater than or equal to 2

> x[x!=2]
# all values not equal to 2
```

If you have a two-dimensional matrix or data frame, you can access its element by separating each index with a comma. The first index is the row number, and the second index is the column number. This can also be used to access higher-order matrices.

```
> y[3,5]
# element in the third row and fifth column
```

In a dataframe where variables (columns) have names, these can be accessed with the dollar sign operator. Brackets can then be used to access individual elements. To find the names of the columns, use the `colnames()` function.

```
> colnames(y)
# this will display the names of the variables in y

> y$d18O
# this will display all elements of the d18O
# variable in y

> y$d18O[1:4]
# only the first four elements of d18O
```

To avoid having to use the $ notation, which can be cumbersome, you can use the **attach()** function to make all of the variables of an array or dataframe accessible. The **detach()** function removes them from this ability. Note that any changes made to a variable while attached are NOT saved to the original variable; in effect, **attach()** creates a copy of the original variable to be used.

```
> d18O
# this won't work; would need to use y$d18O

> attach(y)
```

```
    # make all variables within y directly available without
    # using $ notation

    > d18O
    # this now displays the values of this variable

    > detach(y)
    # removes all variables of y from this type of access
```

Vectors can be sorted easily with the **sort()** function

```
    > sort(x)
    # will sort x into ascending order, by default

    > sort(x, decreasing=TRUE)
    # now into descending order

    > rev(x)
    # this simply flips the order of the elements,
    # such that the first element is now the last,
    # and so on

    > rev(sort(x)) # another way to sort into descending order
```

Sorting a matrix or data frame is more complicated. The following would sort the dataframe
A by the values in column 7 and store the result in dataframe B.

```
    > B <- A[c(order(A[,7])),]
```

R also includes a type of object, known as a list, which can contain almost anything. Many
functions that return more than a single value, such as a regression, return a list that contains
a variety of components. In the case of regression, these might include the slope, the inter-
cept, confidence limits, p-values, residuals, etc. Elements of a list are commonly named and
they can be accessed by this name or by their numerical position. The names of the elements
can be found with the **names()** command. If the element of a list is a vector or a matrix,
subscripts can then be used to access the individuals elements of that vector or matrix. The
attach() and **detach()** functions also work with lists.

```
    > names(regressionResults)
    # displays the names of the regressionResults list

    > regressionsResults$residuals
    # displays the residuals element of a regression

    > regressionResults[[2]]
    # displays the residuals by their position in the list

    > regressionResults[[2]][3]
    # displays the value of the third residual
```

```
> regressionResults[[2]][abs(regressionResults[[2]])>1]
# displays all residuals greater than one
```

Generating Numbers

Writing long, regular sequences of numbers is tedious and error-prone. R can generate these quickly with the **seq()** function.

```
> seq(from=1, to=100)
# integers from 1 to 100

> seq(from=1, to=100, by=2)
# odd numbers from 1 to 100

> seq(from=0, to=100, by=5)
# counting by fives
```

Many different types of random numbers can also be generated. A few common examples include **runif()** for uniform or flat distributions, **rnorm()** for normal distributions, and **rexp()** for exponential distributions.

```
> runif(n=10, min=1, max=6)
# generates ten random numbers between 1 and 6,
# assuming a uniform distribution

> rnorm(n=10, mean=5.2, sd=0.7)
# ten normally distributed numbers, with mean of 5.2 and
# standard deviation of 0.7

> rexp(n=10, rate=7)
# ten exponentially distributed numbers, with rate
# parameter of 7
```

Saving and Exporting Results

From the R console window, it is possible to copy and paste the entire R session into a text file to save a complete record of your results, at least all of the commands and those results that were displayed. You will likely also find it useful to save in this way a library of commands and command sequences you've used to do your analyses, which will allow you to replicate your work, to build on it subsequently, and to apply your analyses to other data sets.

The history of commands from a session can be save to a file with the **savehistory()** function and subsequently reloaded with the **loadhistory()** function. It is recommended that you use a distinctive suffix, like .Rhistory, to make the contents of the file clear.

```
> savehistory(file = "myCommands.Rhistory")

> loadhistory(file = "myCommands.Rhistory")
```

In OS X, graphics produced in R can be saved as pdf files by selecting the window containing the plot, and choosing Save or Save As from R's file menu. In Windows, graphics can be saved this way to metafile, postscript, pdf, png, bmp, and jpeg formats.

Alternatively, once graphics in R have been generated, they can be saved as a postscript file, which can be edited in Adobe Illustrator. To save a postscript file, call the **postscript()** function, then the graphics commands needed to generate the plot, and finally **dev.off()** to close the postscript file. While the graphics commands are being run, they will be executed only in the postscript file and will not appear onscreen. A typical strategy is to develop a graph onscreen, then open a postscript file, and finally rerun the complete set of plotting commands to generate a copy of the plot onscreen.

```
> postscript(file="myGraphicsFile.ps", height=7, width=7)

> plot(x,y)
# and any other graphics commands needed

> dev.off()
# to close the myGraphicsFile.ps file
```

Any objects generated in R, such as variables, vectors, arrays, data frames, functions, etc. can be saved to a file by using the command **save.image()**. Although this will not save any graphics, nor the command history, it is useful for saving the results of a series of analyses, so they will not have to be regenerated in future R sessions. Your work can be reloaded with **load()**, allowing you to continue your work where you left off.

```
> save.image(file = "myResults.RData")

> load(file = "myResults.RData")
```

Basic Statistics

R has a series of basic functions for calculating simple descriptive statistics from a vector of data. These include:

```
> mean(x)
# arithmetic mean

> median(x)

> length(x)
# returns number of elements in x, that is, sample size

> range(x)

> sd(x)
# standard deviation
```

```
> var(x)
# variance
```

A broad series of statistical tests are readily available using R. Some of the most common include

```
> t.test(x)
# t-tests for equality of means, with alternatives for
# equal and unequal variances

> var.test(x)
# F-test for equality of variance

> cor(x)
# correlation coefficients, including Pearson, Spearman,
# and Kendall coefficients

> lm(y~x)
# simple least squares regression - there are an enormous
# number of alternative regression possibilities in R

> anova(lm(x~y))
# ANOVA of x with respect to y, where y is a factor

> wilcox.test (x, mu=183)
# one-sample Wilcoxon signed-rank test (Mann-Whitney U test)
# for mean

> kruskal.test(x~y)
# Kruskal-Wallis test, a non-parametric equivalent of
# ANOVA, testing for differences in x as a function of y,
# where y is a factor

> ks.test(x,y)
# Kolmogorov-Smirnov test for equality of distributions
```

R includes a wide range of statistical distributions that can accessed in numerous ways, such as generating random numbers, finding probabilities or p-values, finding critical values, or simply drawing the probability density function. These commands are all set up in a similar way, although their parameters may differ. As an example, the normal distribution can be used these ways with the **rnorm()**, **pnorm()**, **qnorm()**, and **dnorm()** functions.

```
> rnorm(n=10, mean=5, sd=2.2)
# generate ten random numbers from a distribution with a
# mean of 5 and a standard deviation of 2

> pnorm(q=1.96)
# find the area under a standard normal curve from the left
# tail to a value of 1.96. Note that 1 minus this value would
```

```
# correspond to the p-value for a Z-test, where Z = 1.96

> qnorm(p=0.975)
# find the critical value corresponding to a given
# probability.

> dnorm(x=1.5)
# find the height of the probability density function at 1.5
# If this was done over the full range of x, this would
# produce a plot of the shape of the probability function
```

Other distributions, such as the uniform, exponential, and binomial distributions follow a similar syntax.

```
> rbinom(n=10, size=15, prob=0.1)
# generate ten number of successes for binomial trials where
# n is 15 and the probability of success is 0.1

> pexp(q=7.2)
# find the area under a curve for an exponential distribution,
# from the left tail to a value of 7.2

> qunif(p=0.95)
# find the critical value for a uniform distribution for a
# probability of 0.95
```

Plots

When calling an R plotting routine, the plot will automatically be generated in the active plotting window, if one has been generated, or a plotting window will be created with the default settings if no plotting window is currently active. For more control over the plotting window, the window commands can be called. Specific onscreen windows are available for OS X, X11, and Windows systems. Other types of windows can also be generated for particular file types.

```
> quartz(height=7, width=12)
# for a custom window 7" high and 12" wide in OS X

> X11()
# for X11 on UNIX, LINUX, and OS X systems

> windows()
# for Windows

> postscript()
# to generate a postscript file, which can
# be edited in Illustrator
```

```
> pdf()
# to generate a pdf file


> png()
# to generate a bitmap PNG file; not always available


> jpeg()
# to generate a jpg file; not always available
```

Histograms are generated with the **hist()** function, which can be controlled in various ways.

```
> hist(x)
# generate a default histogram


> hist(x, breaks=20)
# suggest twenty separate bars, but note that the actual
# number of bars may differ


> brk <- c(-2.5, -2, -1, 0, 1, 1.5, 2, 2.5)
> hist(x, breaks=brk)
# force specific breaks in the distribution
```

X-Y plots or scatterplots are generated with the **plot()** function.

```
> plot(x, y)
# a quick and dirty x-y plot
```

Labels for the x and y axes can be set with the xlab and ylab parameters. The title can be set with the main parameter.

```
> plot(x,y, main="Iron contamination at mine site", xlab="pH",
  ylab="Fe content")
# specifying the labeling of the plot and the two axes
```
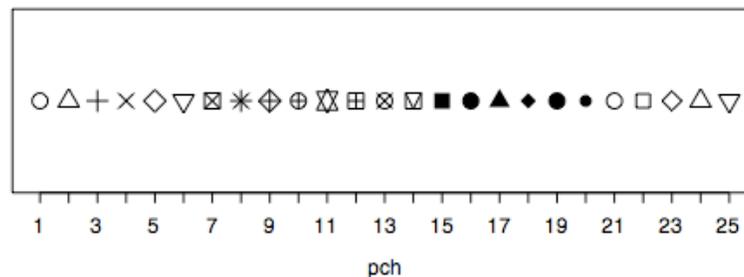
The pch parameter can be set in **plot()** to specify a particular type of symbol.

```
> plot(x, y, pch=16)
# points will be plotted as small filled circles
```

A wide variety of plotting symbols are available by changing pch:

By changing the type parameter, you can control whether lines or points are plotted.

```
> plot(x, y, type="p")
# plot points only

> plot(x, y, type="l")
# plot connecting lines only: note small letter L

> plot(x, y, type="b")
# plot both points and lines

> plot(x, y, type="n")
# plots nothing, just draws axes and labels
```

Symbol colors can be set in the **plot()** or **points()** calls with the col parameter. To see the full list of named colors, use the **colors()** function. Colors can also be specified with by RGB value.

```
> plot(x, y, pch=16, col="green")
# small green filled circles

> colors()
# lists all the named colors

> plot(x, y, col="#00CCFF")
# pale blue, using its RGB value
```

Lines can be added to a plot with the abline command, which is quite flexible

```
> abline(h=5)
# draws a horizontal line at a y-value of 5

> abline(v=10)
# draws a vertical line at a x-value of 10

> abline(a=ycept, b=slope)
# draws a line of given intercept and slope

> abline(lm(y~x))
# draws a line fitted by a regression function - here the
# lm function
```

Equations and special symbols can be used in figure labels and as text added to figures. See the help page for **plotmath()** for more details. **text()** is used for placing additional text onto a figure, **expression()** creates an object of type "expression", and **paste()** is used to concatenate items. The help pages for these functions are useful for seeing the basic ways for combining these useful functions.

```
> plot(x, y, ylab=expression(paste(sqrt(x))))
# makes the y-axis label be the square root of x


> plot(x, y, pch=x, ylab=expression(paste(delta^18, "O")))
# this will make the y-axis label be delta 18 O,
# in correct notation
```

Complex plots are typically built in stages, with groups of points added sequentially, custom axes added separately, text and graphic annotations added. To build a plot in this way, the type will have to specified as "n" to prevent data points from being plotted initially, and axes set to "FALSE" to prevent the axes from being drawn at first.

```
> plot(x, y, type="n", axes=FALSE)
```

Following this, points can be added with the **points()** function.

```
> points(x, y)
# will add these x-y points to an existing plot.  Only those
# points displayable in the current limits of the axes will be
# shown
```

Axes can be added individually with the **axis()** function.

```
> axis(1,at=seq(-1.0, 2.0, by=0.5), labels=c("-1.0", "-0.5",
   "0.0", "0.5", "1.0", "1.5", "2.0"))
# the initial 1 specifies the x axis, the 'at' specifies where
# tick marks should be drawn, and the 'labels' specifies
# the label for each tick mark.
```

Text annotations can be added with the **text()** function.

```
> text(x=10, y=2, labels=expression(sqrt(x)))
# insert the square root of x at the specified x-y
# position on the plot
```

A variety of shapes can also be added to the plot. These can also all be customized with particular line types (dashed, dotted, etc.), colors of lines, and colors of fills. See lty in **par()** for customizing line types. It is worth studying **par()** in detail, because this is one of the main ways that plots can be customized.

```
> rect(xleft=2, ybottom=5, xright=3, ytop=7)
# will draw a box between (2,5), (3,5), (3,7), and (2,7)


> arrows(x0=1, y0=1, x1=3, y1=4)
# will draw an arrow from (1,1) to (3,4)


> segments(x0=1, y0=1, x1=3, y1=4)
# will draw a line segment connecting (1,1) and (3,4)


> lines()
# will draw a series of connected line segments
```

```
> polygon()
# can be used to draw polygons of any shape, not just standard
# symmetrical convex polygons
```

A box can be drawn around the entire plot with the **box()** function.

```
> box()
```

It is possible to have multiple plots drawn together. To do this, use the **par()** function to set the mfrow parameter. Plots will be added to this panel from left to right, starting in the upper left corner.

```
> par(mfrow=(c(3,2)))
# this will allow six plots to be shown simultaneously,
# with three rows of two plots. Following this, the next six
# plot() calls will fill this panel
```

Complex or customized plots are generally built in parts rather than with a single command, such as in this example:

```
> quartz(height=4, width=10)
# make a wide window

> plot(juliandate, SwH, main="Station 46042 - Monterey - 27 NM
   West of Monterey Bay, California", xlab="",
   ylab="Significant Wave Height (m)", axes=FALSE,
   type="n", ylim=c(0,6))
# make plot and label axes, but don't draw axes or ticks,
# don't draw any points, and set range of y values to zero
# to six

> rect(12,0,18,6,col="#EEEEEE",border="#EEEEEE")
# draw a light gray box to emphasize a portion of
# the data

> abline(v=1, col="#AAAAAA")
> abline(v=32, col="#AAAAAA")
# draw two vertical lines to emphasize divisions
# along the x-axis

> points(juliandate, SwH, type="l")
# add points to plot

> axis(1, at=seq(0,46,by=1), labels=FALSE, tcl=-0.3)
# draw closely spaced short ticks along x-axis

> axis(1, at=seq(0,45,by=5), labels=c("30 Nov",
   "5 Dec","10","15","20","25","30","4 Jan","9","14"))
# draw widely spaced longer ticks along x-axis and
```
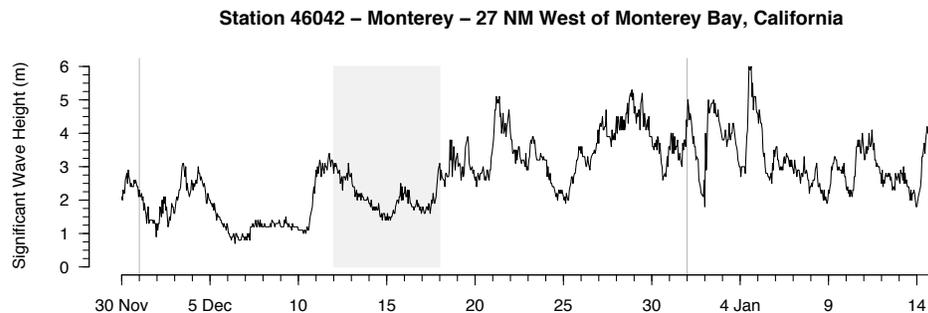
```
# add labels to axis

> axis(2, at=seq(0,6,by=0.25), labels=FALSE, tcl=-0.3)
# draw closely spaced short ticks along y-axis

> axis(2, at=seq(0,6,by=1), labels=
    c("0","1","2","3","4","5","6"), las=1)
# draw widely spaced, longer ticks along y-axis and add labels
```

Station 46042 – Monterey – 27 NM West of Monterey Bay, California

Programming in R

In general, loops are slow in R compared to its vectorized math routines. However, some problems require loops and R has the ability to solve them.

The if loop is a generalized loop like the for loop in C or the do loop in Fortran. Its syntax is intuitive and it is the loop you will use most frequently. Note the use of curly braces if there is more than one statement inside the loop. There are various styles as to where the curly braces may be placed in order to help readability of the code, but they do not alter the execution. In general, indenting the code inside the loop helps the contents loop to be more obvious.

```
> x <- c(1,3,-2,4,8,-12)

# calculate variance and skewness as you would in C or
# Fortran, using loops
> sum <- 0
> for (i in 1:length(x)) sum <- sum + x[i]
> mean <- sum/length(x) # hard way to find the mean
> sumsquares <- 0
> sumcubes <- 0
> for (i in 1:length(x)) {
    sumsquares <- sumsquares + (x[i]-mean)^2
    sumcubes <- sumcubes + (x[i]-mean)^3
```

```
    }
> variance <- sumsquares / (length(x)-1)
> skewness <- sumcubes / ((length(x)-1)*sqrt(variance)^2)

# now, calculate them as you could in R, using vectorized math
# Note that these are not only faster, but much more compact
> variance <- sum((x-mean)^2)/(length(x)-1)
> skewness <- sum((x-mean)^3)/(length(x)-1)*sqrt(variance)^2)
```

In addition, R has looping constructs called repeat and while, as well as the break statement that allows you to escape from a loop and the next statement that lets you jump to the next iteration of a loop.

```
> repeat {expressions}

> while(some condition) {expressions}

> break
# note - no parentheses

> next
# note - no parentheses
```

Customizing R

One of the great advantages of R in being open source is that there are a very large number of contributed packages that can be used to solve a much greater range of problems than in the R base package. For example, one can find packages specifically directed to non-parametric statistics, signal processing, or ecological analyses. Using these involves two steps, installing them on a system, and loading them while in an R session. Most packages can be installed from the R site: http://CRAN.R-project.org/. In OS X, packages can be installed through the package installer under the Packages & Data menu. To check which packages have been installed and are available to be loaded, use the **library()** function.

```
> library()
```

Once a package is installed, it can be loaded by calling it with the **library()** function.

```
> library(vegan)
# loads the vegan package
```

Functions you write can be saved in a text file and then reloaded into an R session with the **source()** function.

```
> source("myfunctions.r")
```

You can control the working directory of R with the **setwd()** function, which allows you to keep all R files associated with a project in one place.

```
> setwd("/Users/sholland/Documents/Rdata")
```

If you find yourself commonly typing the same commands at the beginning or the end of
every R session, you can set R up to automatically execute those commands whenever it starts
or quits by editing the **first()** and **last()** functions in your .Rprofile file. On OS X and
other UNIX/LINUX systems, the .Rprofile file is an invisible file that is located at the root
level of a user's home directory. Within that file, assuming it exists, is where the .First and
.Last functions are located. Common commands to place in the .First file include loading any
commonly used libraries or data sets, setting the working directory, setting preferred graphics
parameters, and loading any files of R functions. For example, this is a sample contents of a
.First function.

```
.First <- function() {
library(utils)
library(grDevices)
library(graphics)
library(stats)
library(vegan)
library(MASS)
library(cluster)
setwd("/Users/sholland/Documents/Rdata")
source("myfunctions.r") }
```