A SHORT R TUTORIAL

Steven M. Holland

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

August 2013

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. The Manuals link on the left side of the R home page leads to a series of resources, such as *R Administration and Installation*, which gives 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 disk image 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 read me 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. A read me file contains the installation instructions.

Learning R

The number of books on R is simply exploding. Several that I've had a chance to read are listed below, from the more basic to the more advanced. All attempt to teach both R and statistics, with varying 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 numerous manuals and tutorials available for R online, and a Google search will provide you with plenty to read. The manuals link on the R home page has links to three important guides. The *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) that comprehensively lists all help files. This can also be purchased for a modest cost from amazon.com, but the help function in R is the easiest and cheapest way to get help.

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, type the name of the variable at the prompt.

```
> x
```

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

```
> x <- 3 + 4
> x <- 2 - 9
> x <- 12 * 8
> x <- 16 / 4
> x <- 2 ^ 3
```

Comments are always preceded by a # sign, and what follows on that line will not be executed.

```
# x <- 2 * 5; everything on this line is a comment
# none of this will execute

> x <- 2 * 5 # comments can be placed after a statement
```

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

```
> x<-3+7/14
> x <- 3 + 7 / 14
# both lines produce the same result
```

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. This will show the value of X if X has been used
# before, a problem if 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. The down arrow lets you move towards more recent commands.

Functions

R has a rich set of functions, which are called with any arguments in parentheses. Functions are also objects. The maximum of two values would be calculated with the **max** function:

```
> max(9, 5)
> max(x, 4) # objects can be inserted into functions
```

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

```
> objects()

> objects
# this returns an error, since the parentheses are missing
```

Functions usually return a value. If the function 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 displays the maximum of 9 and 5 on the screen

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

Multiple functions can be called in a single line, and functions can be nested, with the result of one function used as an argument for another function. Nesting functions too deeply can make the command long, hard to interpret, and hard to debug if something should go wrong.

```
> y <- 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. Use the help pages.

```
> help(max)
> ?max
> ? max
# all three will get help for the max() function
```

You can write your own functions. For example, the function below raises a base to an exponent. In parentheses after the word **function** is a list of arguments to the function, that is, values that must be input into the function. The contents of the function, that is, the commands the function performs, are enclosed in curly braces. The indentation of these statements is not required, but it makes the function easier to read.

```
> pow <- function(base, exponent) {
      result <- base^exponent
      result
}
```

The arguments of a function can be called in two ways, by name and by position. When you call arguments by name, they can be listed in any order and the function will give the same result:

```
> pow(base=3, exponent=2)
> pow(exponent=2, base=3) # same result
```

Calling arguments by position saves typing by hiding the argument names, but the values must in the correct order.

```
> pow(3,2)      # returns 3^2, or 9
> pow(2,3)      # returns 2^3, or 8
```

In both of these statements, the first position is assumed to be the first argument in the definition (**base**) and the second position is assumed to be the second argument (**exponent**).

Some functions have default values for some arguments. For example, the pow function could be written as

```
> pow <- function(base, exponent=2) {
      result <- base^exponent
      result
}
```

which makes the default exponent equal to 2. If you write **pow(3)**, you'll get 9 in return. If you want a different exponent, specify that argument, such as **pow(3, 3)**, which would produce 27. Refer to a function's help page to see which arguments have default assignments.

Several functions are useful for manipulating objects in R. To show all current objects, use **`objects()`** and **`ls()`**. To remove objects, use **`remove()`**.

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

> ls()
# also displays the names of all 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. 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 third type of object, called a vector, which you will use frequently. A vector is a list of values, which 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 **`x`** 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 this, the **x** matrix would be:

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

For most larger data sets, it is much easier to enter the data in a spreadsheet (like Excel), check it for errors, and then export the file into a format you can read into R, such as comma-delimited or tab-delimited text. In Excel, choose Save As..., then select "Comma-separated values (.csv)" or "Tab-delimited text (.txt)". Both are text-only formats easily read by almost any program, including R.

To read files into R, you need to know the path to the file, which you can specify as you read in the file or by specifying the working directory first. Usually, you'll want to specify the working directory and keep all your files in one place for that R session. Whenever you read or write files, R will work in your current directory unless you specify another path to the file.

The working directory of R is set with the **setwd()** function:

```
> setwd("/Users/myUserName/Documents/Rdata")
# UNIX (OS X and Linux)

> setwd("C:\\Documents and Settings\\myUserName\\Desktop")
# Windows
```

To see the directory you are currently using, use the **getwd()** function:

```
> getwd()
```

Once you've set your working directory, you can read a vector into R with the **scan()** function. Be sure to put the file and path in quotes; single quotes and double quotes both work, but be consistent.

```
> myData <- scan(file="myDataFile.txt")
# single quotes (') work, too
```

If you want to access a file that's not in your current directory, you will need to include the path:

```
> myData <-
  scan(file="/Users/myUserName/Documents/myDataFile.txt")
# a path for UNIX systems (Linux, OS X)

> myData <- (file= 'C:\\SomeDirectory\\myDataFile.txt')
# a path for Windows
```

To read a matrix into R and save it as a data frame (another common R object, like a matrix, but the columns can be of different types: numeric, strings, boolean, etc.), use the **read.table()** function. This function assumes that your matrix is set up in the correct format, with columns corresponding to variables and rows corresponding to samples. All cells

should contain a value; don't leave any empty. If a value is missing, enter **NA** in that cell. Variable names can be long (at least 34 characters) and do not need to be single words; any blank spaces in a variable name will be replaced with a period in R when you import the table.

Although there are many options for **read.table()** (see the help page), you will primarily pay close attention to three things, which you can determine by examining the data file in a spreadsheet or text editor. First, check to see if there are column names for the variables, that is, whether a *header* is present. Second, check to see if there are row names and what column they are in (usually the first column). Third, check to see what separates the columns; tabs or commas are the most common delimiters.

For example, the following would assume that the first row of a data file (**myDataFile.txt**) 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=","**).

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

If there is no header, leave out the **header** argument, because the default is that a header will be absent. Likewise, if there are no row names, leave out the **row.names** argument. If the values are separated by tabs, use **sep="\t"**.

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; if you skip it, your changes
# will be discarded.

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

Accessing Data

Elements of a vector can be called by using subscripts. To see the first element of x, type:

```
> x <- c(3, 7, -8, 10, 15, -9, 8, 2, -5, 7, 8, 9, -2, -4, -1)
> 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)]
# returns the first, third, fifth, and seventh elements
# Be sure to include the c(), or this will produce an error,
# because it thinks you are accessing a four-dimensional array

> x[3:10]
# returns the 3rd through 10th elements

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

You can use conditional logic to select elements meeting certain criteria

```
> x[x>0]
# will display all positive values in x

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

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

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

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

The way these work is that the conditional statement generates a vector of TRUE and FALSE values, which are used to select the elements of the vector to return.

If you have a two-dimensional matrix or data frame, you can access its elements by separating each index with a comma. The first index is the row number, and the second is the column. This can also be used to retrieve elements of 3-dimensional or even higher-order matrices.

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

In a data frame where variables (columns) have names, these can be accessed with the **$** operator. Brackets are used to access individual elements. To find the names of the columns, use the **colnames()** function.

```
> colnames(y)
# display the names of the variables in the y data frame

> y$d180
```

```
# display all elements of the d18O variable in the y data
# frame

> 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 data frame accessible. The **detach()** function stops them from being accessible this way. Any changes made to a variable while it is attached are *not* saved to the original variable; **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

> d18O
# no longer works
```

Vectors are sorted with the **sort()** function

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

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

> rev(x)
# flip the order of the elements, so that the first is now
# last, and vice versa

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

Sorting a matrix or data frame is more complicated. The following would sort the data frame A by the values in column 7 and store the result in data frame 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 values between 1 and 6,
# assuming a uniform distribution

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

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

Basic Statistics

R has functions for calculating simple descriptive statistics from a vector of data, such as:

```
> mean(x)    # arithmetic mean
> median(x)  # middle value
> length(x)  # number of elements in x (sample size)
> range(x)   # largest and smallest value
> 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, y)             # t-test for equality of means
> var.test(x, y)           # F-test for equality of variance
> cor(x, y)                # correlation coefficient
> lm(y~x)                  # least squares regression
> anova(lm(x~y))           # ANOVA (analysis of variance)
> wilcox.test (x, mu=183) # Mann-Whitney U test
> kruskal.test(x~y)        # Non-parametric ANOVA
> ks.test(x,y)             # Kolmogorov-Smirnov test
```

R includes a wide range of statistical distributions that can used to generate random numbers, find probabilities or p-values, find critical values, or draw probability density functions. These commands are all set up in a similar way, although their arguments may differ. For example, the normal distribution can be used with the `rnorm()`, `pnorm()`, `qnorm()`, and `dnorm()` functions.

```
> rnorm(n=10, mean=5, sd=2.2)
# generate ten random numbers from a normal 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 the number of successes for 10 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
```

One of the great strengths of R is the incredibly large number of available statistical analyses. Many of these are built into R, but an ever-growing number of them are freely available in user-contributed packages. There's a good chance that if you need to do an analysis, it is already available in R.

Plots

The other great strength of R is the quality of its plotting routines. Many types of plots are available, and quite complex plots can be produced. R's plots are far superior to those made in spreadsheet programs like Excel.

When calling an R plotting function, the plot will automatically be generated in the active plotting window, if one already exists, or, if none exists, a new plotting window will be created with the default settings. For greater 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 OS X
> X11()                      # For UNIX and LINUX
> windows()                  # For Windows
```

Histograms or frequency distributions 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
```

Scatterplots (x-y plots) are generated with the **plot()** function.

```
> plot(x, y)
# argument one is the horizontal axis, argument two is the
# vertical axis
```

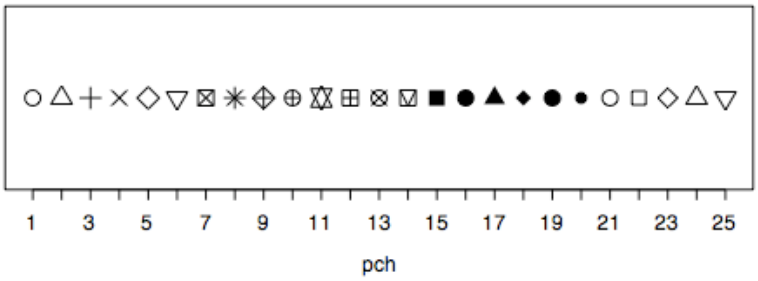
Labels for the x and y axes can be set with the **xlab** and **ylab** arguments. The title can be set with the **main** argument.

```
> plot(x,y, main="Iron contamination at mine site", xlab="pH",
 ylab="Fe content")
```

The **pch** argument can be set in **plot()** to specify a particular type of symbol.

```
> plot(x, y, pch=16)  # pch 16 is a small filled circle
```

Twenty-five plotting symbols are available by through **pch**:



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

```
> plot(x, y, type="p") # points only
> plot(x, y, type="l") # connecting lines only, no points
> plot(x, y, type="b") # both points and lines
> plot(x, y, type="n") # no points, only axes and labels
```

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

```
> plot(x, y, pch=16, col="green") # small green filled circles
> colors() # lists all named colors
> plot(x, y, col="#00CCFF") # 00CCFF is RGB for pale blue
```

Lines can be added to a plot with the **abline()** function, which is quite flexible

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

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

```
> abline(a=intercept, b=slope)
# line with a specified intercept and slope

> abline(lm(y~x))
# line fitted by the lm regression 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. The function **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, 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, with text and graphic annotations added. To build a plot this way, set **type="n"** to prevent data points from being plotted, and set **axes=FALSE** to prevent the axes from being drawn.

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

Points can be added to the current plot with the **points()** function. Only those points that are visible within the current limits of the axes will be shown.

```
> points(x, y)
```

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 argument specifies the axis to be drawn, with 1 indicating the x axis and 2 indicating the y-axis. The **at** argument specifies where the tick marks should be drawn, and the **labels** argument specifies the label to be displayed next to each tick mark.

Text annotations can be added with the **text()** function, where the **x** and **y** arguments specify the position of the annotation, and the **labels** argument specifies what text will be added.

```
> text(x=10, y=2, labels=expression(sqrt(x)))
```

Shapes can also be added to the plot, and these can 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)
# draws a box between (2,5), (3,5), (3,7), and (2,7)

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

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

> lines()
# draws 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()
```

Multiple plots can be drawn together by using the **par()** function and setting the **mfrow** argument. Each successive call to **plot()** will fill the panel from left to right, starting in the upper left corner.

```
> par(mfrow=(c(3,2))) # 6 plots, with 3 rows of 2 plots
```

Complex or customized plots are generally built in parts rather than with a single command, such as in the following example. Note that this example was preceded by experimenting with different parts of the plot to determine the order in which these commands would be given. Plotting begins by making a window of an appropriate aspect ratio:

```
> quartz(height=4, width=10)
```

Once the data is read in and attached for convenience, **plot()** is called to specify the x and y data, the titles, the axis labels, and the limits of the y data range (ylim). The type is set to n, because the data will be added subsequently.

```
> waves <- read.table(file="waves2.txt", header=TRUE, sep=",")
> attach(waves)
> 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))
```

Next, a light gray rectangle is added to emphasize a particular window of the data. This is added first, so that the data will be added on top of the rectangle. If the data were drawn first, this rectangle would obscure the data.

```
> rect(12,0,18,6, col="#EEEEEE", border="#EEEEEE")
```

Vertical lines are added to emphasize divisions of the data along the x-axis:

```
> abline(v=1, col="#AAAAAA")
> abline(v=32, col="#AAAAAA")
```

The data is added as a series of lines that connect individual values, rather than showing dots or some symbol for each data point.

```
> points(juliandate, SwH, type="l")
```

The x-axis will be built in two stages. First, closely spaced short ticks are added:

```
> axis(1, at=seq(0,46,by=1), labels=FALSE, tcl=-0.3)
```

Next, widely spaced longer ticks are added, and these are labelled:

```
> axis(1, at=seq(0,45,by=5), labels=c("30 Nov",
  "5 Dec","10","15","20","25","30","4 Jan","9","14"))
```

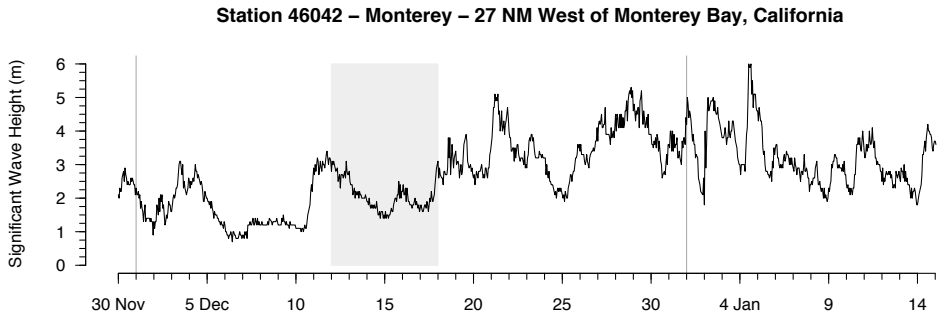
Likewise, the y-axis is built in two stages, beginning with closely spaced short ticks:

```
> axis(2, at=seq(0,6,by=0.25), labels=FALSE, tcl=-0.3)
```

These are followed with widely spaced long ticks with labels:

```
> axis(2, at=seq(0,6,by=1), labels=
  c("0","1","2","3","4","5","6"), las=1)
```

This is the final plot.



Saving and Exporting Results

From the R console window, you can copy and paste the entire R session into a text file to save a record of the commands you issued and the results that were displayed. This is a useful way to save a set of commands you've used to do your analyses. Making a habit of this allows

you to replicate your work, to build on it subsequently, and to apply your analyses to other data sets.

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.

Graphics can also be built and saved to a file from the command line. To open a pdf file, which can be edited in Adobe Illustrator, use the **`pdf()`** function. After this, run all of your plotting commands, then call **`dev.off()`** to close the pdf file. Note that your plot will not be displayed while you do this, although it is being written to a file. A typical strategy is to develop a plot onscreen, then open a pdf file, and finally rerun the complete set of plotting commands to generate and save a copy of the plot.

```
> pdf(file="myPlot.pdf", height=7, width=7) # open a file
> plot(x,y) # and any other graphics commands needed
> dev.off() # close the file
```

You can also plot to other types of graphics files. Again, remember to close the file with **`dev.off()`**.

```
> postscript()
> png()
> jpeg()
```

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. If your work involves long-running computations to build an object, **`save.image()`** will allow you to save and quit your work, then reopen the file and pick up where you left off. 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")
```

Programming in R

If you have previous programming experience, you will be tempted to program in R with loops. Don't. Calculations in R are vectorized, meaning that operations can take place simultaneously on all elements of a vector. In comparison, loops are quite slow in R and should be avoided whenever possible. Some problems require loops and R can use them.

The **`for`** loop is a generalized loop like the *for* loop in C or the *do* loop in FORTRAN. If you have programmed before, you will find the syntax intuitive, and you will find 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 to help readability of the code, but they do not alter the execution. Indenting the code inside the

loop helps to make the contents of the loop more apparent, and although indenting isn't required, it is good form to do it.

```
> 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)
```

R has an **if** test to let you run one or more statements if some condition is true.

```
> if (x < 3) y <- a + b
```

There is also an **else** test, and you can execute multiple statements by enclosing them in brackets.

```
> if (x < 3) {
  y <- a + b
  z <- sqrt(x)
 } else {
  y <- a - b
  z <- log(a + b)
 }
```

R has a **while** loop to let you execute a block of code for as long as some condition is true.

```
> while(x >= 0) {some expressions}
```

Note that if this condition never becomes false, this will generate an infinite loop. R also has a repeat loop that is also an infinite loop unless you insert a **break** statement some. Usually you would use a repeat loop, but test for some condition inside the loop that would trigger the **break** statement to get you out of the loop. Note that the **break** statement is not a function and is not followed by parentheses.

Customizing R

Packages

Another great advantages to the open-source nature of R is that users have contributed an astonishing number of packages that can be used to solve a much greater range of problems than with just the R base package. For example, there are packages specifically directed to non-parametric statistics, signal processing, and ecological analyses. To use these packages, you must first *install* them on your system, and then *load* 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)    # load the vegan package
```

A package need be installed only once, but it needs to be loaded every R session in which you wish to use it.

Function files

As you become familiar with R, you will write your own functions. To have easy access to these, save your functions to a text file. These can be loaded into an R session with the **`source()`** function.

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

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 arguments, and loading any files of R functions. For example, these are sample contents of a .First function.

```
.First <- function() {
library(utils)
library(grDevices)
library(graphics)
library(stats)
library(vegan)
library(MASS)
```

```
library(cluster)
setwd("/Users/myUserName/Documents/Rdata")
source("myfunctions.r") }
```

This function loads a series of libraries that I commonly use, switches me to the directory I most commonly work from, and loads a set of functions that I've created. By putting all of this into the .First function, I save myself typing this every time I start R.

Web resources

Everyone likely has their favorite web sites for R, and these are mine.

R-bloggers (http://www.r-bloggers.com) is a good news and tutorials site that aggregates from over 450 contributors. Following its RSS feed is a good way to stay on top of what's new and to discover new tips and analyses.

Cookbook for R (http://www.cookbook-r.com) has recipes for analyzing your data.

Stack Overflow (http://stackoverflow.com/questions/tagged/r) is a question and answer site for programmers. Users post questions, other users post answers, and these get voted up or down, so you can see what the community regards as the right answer. Stack Overflow is great for many languages, and the R community that uses it is growing.

Finally, remember if you run into problems that **Google is your friend**.