# **BASICS OF R**

## A Primer

Maestro playing on computer keyboard



Don Edwards

Department of Statistics

University of South Carolina

Columbia, SC 29208

[*edwards@stat.sc.edu*](mailto:edwards@stat.sc.edu)

September 2004

Edited July 2007, August 2009, September 2011, May 2012

Revised 2015, 2017, 2020

# 1. INTRODUCTION

## 1.1 Background

R is an implementation and enhancement of the S language, which was developed at AT&T Bell Labs in the 1970s and ‘80s and left in the public domain after the breakup of AT&T. R has many similarities with Splus®, another implementation of S for sale as a commercial software package. R is faster, though, and has an enormous archive of user-written functions and data sets (CRAN = Comprehensive R Archive Network). And, R is free !! …to download it, go to the [CRAN website](http://www.r-project.org/) and follow the directions. R has taken the academic Statistics world by storm; see the [New York Times article](http://www.nytimes.com/2009/01/07/technology/business-computing/07program.html?_r=2&ref=technology&). R is part of the free software movement; if you would like to know more about this, go to the [GNU website](http://www.gnu.org/).

This primer is aimed at an individual with minimal R experience, to introduce the structure and syntax of the language, to provide some basic tools for manipulating and displaying data, and to introduce many other topics which deserve further study. It assumes you are using the Windows implementation of R, version 3.0.2 or later. The implementation for Macintosh is similar in syntax and operation to the Windows implementation. Implementations for Unix, and Linux are also available, and supposedly do not differ dramatically in syntax or operation from the Windows implementation. Differences in R between Windows and other platforms will be noted as they arise. Any suggestions for improvements to this primer can be sent to Don Edwards ([edwards@stat.sc.edu](mailto:edwards@stat.sc.edu)) or John Grego (grego@stat.sc.edu)..

Almost all student users prefer to use R through the environment known as RStudio, which is also free. Select [this link](http://www.rstudio.com/), choose a version and operating system, and download it—just remember you will also need to have R installed. This primer is based on RStudio, since students prefer its more modern IDE (Interactive Development Environment).

R is a powerful language, but it has a learning curve. It is command-driven and case sensitive, unforgiving of careless mistakes, and its error messages leave a bit to be desired. On the other hand, it is not difficult to study R objects as they are created; this is a great help in understanding coding errors. Patience and regular use are keys to learning R. As the cover cartoon suggests, R is like a musical instrument - if you can get past the learning curve, it flows naturally and is ultimately…(gasp)… enjoyable to use.

In this primer, R objects, operators, etc. are written in Courier New font (like this). Syntactically rigid commands, keywords, and built-in function names in R are written in **boldface** to distinguish them from created objects, whose names would usually be chosen by the user. Also, silly generic names like my.object are often used when referring to user-created objects.

Object names can be essentially any length, but must start with an alphabetic character and contain no blanks or special characters. If a name consists of multiple words, usually we connect these with periods. Try not to use names that might be natural choices for an existing built-in R function - some names to avoid, specifically, are “plot”, “t”, “c”, “df”, “T”, and “F”, though these should only cause warning messages or confusion unless you use them as names for functions you create. Choose meaningful names for objects since created objects tend to collect in your workspace, and if they are not well named you’ll forget what they are. It is helpful if names are short, as well – less typing.

To start RStudio from an empty workspace, double-click on the blue RStudio icon on the desktop or under Programs in the Start menu. (or under Applications if you are on a Mac). Start RStudio now; four different frames will appear. They have a lot of functionality, but basically, you will use the upper left frame for coding, the lower left frame for executing code, the upper right frame for managing workspace, and the lower right frame for viewing graphs and R help. In the lower left frame, the Console window should open, and you should see the R command prompt “>“. When you type a command at the prompt, push Enter or Return to execute it. You can page back through recent commands using the up- and down- arrow keys. When you opened RStudio by clicking on the program icon, you were provided with a workspace to store objects that you create. To look at the names of objects in your workspace, call the ls function:

> ls()

You should see the word character(0), which means there is nothing in your workspace yet. You can also execute this command by entering ls() in the upper left window (no command prompt is necessary), highlighting it, then pushing Command-Enter; results will appear in the Console window.

Even though your workspace is empty, save your R workspace immediately in a well-named folder (or directory) for the current project. For example, suppose you have a folder called “stat540” on a USB drive addressed as drive M by your computer. In the upper right-hand frame, select the Environment tab, then select the Save icon (a floppy disk—whatever that was). In the new window, enter a filename in the Save As box (e.g., FirstSession.RData), and then select your USB drive folder to save your workspace. The default file extension for R workspaces is .RData, but it doesn’t hurt to list the extension just in case.

Incidentally, you can specify a folder as your default workspace by selecting Session>Set Working Directory>Choose Directory. By doing so, you do not have to provide long file names when, e.g., reading data files into R.

## 1.2 Your First R Session

This first chapter will be (hopefully) a nonthreatening superficial warmup to R--an examination of a data set giving heights (in inches) of New York Choral Society singers (source: Cleveland, William S. (1993) Visualizing Data. Hobart Press, Summit, New Jersey). Suppose the data are stored in a .csv file (comma-separated-values, a standard format for data) in the folder stat540, which you have set as the working directory. The data file is named NYsingers.csv. Before reading the data, open the data with Excel or similar spreadsheet software and make sure the column (variable) names are meaningful yet easy to type. In this case they are “PitchGroup” and “Height”…could be better but we’ll live with those.

Remember that all these commands could be entered in the upper left-hand frame and executed by highlighting them then pushing Command-Enter. I would recommend clicking in that frame, then selecting “+” in the upper left-hand corner, then clicking New R Script; a blank window will appear. Commands can then be typed in the new window and saved by selecting the Save icon.

Read the data into your workspace with a command similar to this:

> NYsingers.dfr = read.csv(“NYsingers.csv”)

Notice the use of the .csv extension; if something is mistyped, R will say there is “no such file” or make a similar comment. If all goes well, however, the next R prompt should return without a message. To make sure you got the data, check your workspace contents again:

> ls()

You should now see the word “NYsingers.dfr”. You can also check the Environment window in the upper right-hand frame; it will show up under the Data subheading; click on the arrow next to its name , and each variable will be listed, along with their length and first few values. The upper right-hand frame, incidentally, is one of the conveniences of using RStudio rather than R; it is much easier to casually inspect your workspace in RStudio through the Environment tab.

Let’s examine this data frame. One way to examine any object is to simply enter its name at the prompt:

> NYsingers.dfr

And we see that this data frame has two columns and 235 rows (of course, you can learn this more easily by inspecting NYsingers.dfr in the Environment window). We could have found this out more directly by asking specific questions about the data frame:

> dim(NYsingers.dfr)

> names(NYsingers.dfr)

or a single command that accomplishes both of the above and more:

> attributes(NYsingers.dfr)

We will work with this data by applying R functions to its variables. For example, we could calculate the mean singer Height with the command

> mean(NYsingers.dfr$Height)

Or we could make a frequency table showing the number of singers in each PitchGroup using

> table(NYsingers.dfr$PitchGroup)

Notice that technically each variable in the data frame has a two-level name – the data-frame-name and variable name, with $ in between. It would be much easier to work quickly without all this typing…this can be accomplished by “attaching” the data frame:

> attach(NYsingers.dfr)

Now the variables can be referenced using only their variable names. More details on “attach” later.

Try that last command again after attaching the data frame: examine the nominal variable PitchGroup with a frequency table:

> table(PitchGroup)

So there are 35 singers in the first alto group, 21 in the first tenor group, etc. Note that if you forget to attach the data frame, the above command will not work - R will not be able to find PitchGroup.

The above command is an expression which calculates and displays the frequency table, but does not save the results for later use. If you want to use the created table for later use, modify the statement to be an assignment statement, and make up a good name for the result:

> PGtable = table(PitchGroup)

Now look at your workspace again using ls()and look at your newly created object PGtable by entering its name at the prompt.

Make a draft pie chart and barplot for the PitchGroup variable:

**> pie(**PGtable)

> barplot(PGtable)

If you want to use the bar chart in a report, you may want to specify a color and add a title, and you may need to resize it to show all group labels.

> barplot(PGtable,col=”darkred”,xlab=”Pitch Group”)

> title(“NY Choral Society singers by pitch group”)

To see the overwhelming array of color choices, enter

> colors()

To see the color names mapped to the colors themselves, go to <http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf> (look beyond the first page).

Once you get the barplot the way you like it, click in the Plots window to activate it, then go to Export… and either copy it to the clipboard for immediate pasting or save it as a graphics file or PDF file for inserting into documents later.

The variable Height is a quantitative variable. We may want to calculate descriptive statistics on such a variable, for example the mean, variance, standard deviation, maximum, minimum, median, and interquartile range (IQR):

> mean(Height)

> var(Height)

> sd(Height)

> max(Height)

> min(Height)

> median(Height)

> IQR(Height)

The R function range() does not calculate the range of a variable, but rather calculates both the max and min:

> range(Height)

If we want to calculate range as we define it, just do some arithmetic on the max and min:

> max(Height)-min(Height)

The quantile() function by default calculates the “five number summary” for the variable - the min, max, and quartile points:

> quantile(Height)

We can ask for other quantiles / percentiles by requesting these using an optional argument to the function called probs:

> quantile(Height,probs=c(.1,.2,.3,.4,.5,.6,.7,.8,.9))

This gives the 10th, 20th, …, 90th percentiles of the Height variable. Here is an easier way:

> quantile(Height,probs=(1:9)/10)

Of course, since we did not assign names to any of the above descriptive statistics, they have not been saved to the workspace.

R has an impressive array of functions for single-variable graphical descriptives…a stemplot appears in the Console window:

> stem(Height)

Here is an example call to make a histogram in Dodger Blue, specifying the number of class intervals, making the default title blank so a better title can be added later, and demonstrating change of axis labels:

> hist(Height,breaks=20,col=”dodgerblue”,main=””,

+ xlab=”height in inches”,ylab=”frequency”)

In an intermediate or graduate level statistics class, you may learn about smoothed histograms (also called “density estimates”) and empirical cdf functions. Here are calls to compute these in R; the second call shows how we can control the amount of smoothing in the smoothed histogram (I prefer the default here – the second one seems oversmoothed):

> plot((density(Height))

> plot((density(Height,bw=2))

> plot(ecdf(Height))

Finally, here is a boxplot of the entire data set’s Height values:

> boxplot(Height)

Not very interesting…the real value of the boxplot is for side-by-side comparison of a variable across different groups. This can be accomplished here by the following command, with a little added color:

> boxplot(split(Height,PitchGroup),col=”salmon3”)

This plot suggests a research hypothesis - that lower-voiced singers tend to be taller – even within gender. Label this plot a little more completely for presentation:

> boxplot(split(Height,PitchGroup),col=”salmon3”,

+ xlab=”Pitch Group”,ylab=”Height (in)”)

> title(main="NY Choral Society 19?? Heights",

+ sub="Data Source: yadda yadda")

When you need help on a particular topic, for example to attempt to unlock the mysteries of some function, either enter

> help(topic)

or more easily

> ?topic

Or, select the Help window in the lower right-hand frame and enter the topic in the far right-hand search window.

For example, look at the help file for the quantile() function:

> ?quantile

If you cannot quite remember the exact topic name, you could try (for example)

> ??(mybestguess)

Reading R help files takes practice…

This is plenty for one day; when you want to quit R, first make sure you have saved or copied all graphs as needed, and copied your R commands into a script window to be used, e.g., as an Appendix to your report. Any files that you have attached along the way will be automatically detached when you quit R, but if you want to detach them before quitting, simply enter

> detach(2)

or, use the data frame name

> detach(NYsingers.dfr)

Clean up your workspace before you quit; look at its objects and rename these as needed, then remove unwanted junk:

> ls()

> newname = junkobj1

> rm(junkobj1,junkobj2, ...)

If you wish to remove all objects in the workspace, type

> rm(list=ls())

To quit, either select Quit Session under File, or enter:

> q()

# 2. OBJECTS, MODES, AND CLASSES

R is an “object oriented” interactive language. In my non-CSCE way of thinking this means that data are organized into specialized structures called “objects”, of different types. Functions in R are programs which do work, creating new objects from existing ones, making graphical displays, and so on…though functions are also objects.

Try to keep track of your object types as you work, because many functions accept only certain types of objects as arguments, or they do different work depending on what kind of object(s) you use as argument(s) for the function. For example, the following call to the built-in function plot :

> plot(my.object)

will have different consequences depending on whether my.object is a vector, a matrix, a factor, a time series object, etc; these terms will be explained below. When you get comfortable with R, you can (and should) write your own functions as the need arises. You can also call Fortran and C programs from R.

The most common object types for our purposes, in order of increasing complexity, are vectors, factors, matrices, data frames, lists, and functions. Each of these will be discussed in some detail now.

## 2.1 Vectors (and data modes, and assignments)

Vectors are strings of data values, all of the same “mode”. The major data modes are numeric (which is actually an amalgam of two distinct modes, namely integer and double precision floating point), character, and logical (a datum of logical mode has either the value TRUE or the value FALSE, sometimes abbreviated as T and F). Other data modes include complex and NULL.

To examine some vectors, go to the class website and find the file called monarchDGE.RData

This is an R workspace with data already loaded, created from a similar one used at N.C. State University. After saving it in a good location on your computer, click on its R icon to open it. Then type

> ls()

to see the names of the objects already loaded in this workspace; you should see the names Mgroup, Mname, and yrslived in quotes.

Let’s examine these objects. Enter the name of the second object:

> Mname

R will respond by listing out this vector’s values on the screen. To save space, a vector’s values are printed in rows across the screen, with the index number for the first element of each row provided in brackets at the beginning of the row.

The double-quotes on each value in Mname tell us that this vector is of mode character; if we weren’t sure, we could ask (try it):

> mode(Mname)

In this case the character strings are the names of certain past U.S. Presidents, Popes, and English Kings and Queens (so calling them all “monarchs” is not entirely accurate).

We can access an individual value in a vector by specifying its position in square brackets like so:

> Mname[33]

The command above is actually an R expressions that produce a single-element vector of mode character, though it does not save it.

Notice that there are two identical names “Harrison”…this could cause problems. Let’s fix this by assigning new character values to the two locations (be careful to type exactly as shown):

> Mname[9] = “W.H.Harrison”

> Mname[23] = “B.Harrison”

Now look at the vector Mname again; if you are confident that the change is correct, save the workspace.

These last two statement are, of course, assignment statements. The preferred method for assignment used to be a left-pointing-arrow: the “<” character followed by the “-“ character, instead of the “=” sign. You can still do assignment statements with the <- arrow (try it). The arrow is clunky but is much more descriptive of what actually happens in an assignment: the expression on the right-hand-side is evaluated (if possible) using current values of all its objects; the result is then stored in the workspace with the name provided on the left-hand-side. The equal sign is more typical of computer languages and has replaced the left-pointing arrow these days, and will be used in the remainder of this primer. Occasionally you will see a stray “<-“ here, or in an old R message or help file.

Now look at another vector in this workspace, yrslived, by typing its name at the prompt:

> yrslived

This is a numeric vector – the number of years each individual lived after they were “coronated”, i.e. after they assumed responsibilities of president, pope, king, or queen. These values appear to be integers (whole numbers {0,1,2,3,…}) on screen but in this case are actually stored as double-precision floating point real numbers. Confirm this by asking:

> mode(yrslived)

This is not helpful, since “numeric” includes both integer and double precision floating point as sub-modes. Let’s ask directly:

> is.integer(yrslived)

> is.double(yrslived)

> is(yrslived)

And we see that the values of yrslived , though they print on the screen as whole numbers, are in fact internally stored as double-precision floating point values. Double precision values require more memory for storage (8 bytes = 64 bits, to be precise) but can handle very large numbers more accurately than integer-mode values. The third command identifies all data types by which yrs.lived could be classified.

Note that vectors in R differ from vectors in matrix algebra in that they have no orientation: an R vector is neither a row vector nor a column vector. This can cause problems when doing matrix arithmetic if one is not careful. A scalar constant in R is a vector of length 1.

Let’s learn about a few more built-in functions and operators. The function length() for vectors is self-explanatory. Try it by entering

> length(Mname)

The built-in function **c()** combines values (or vectors, or lists), of the same mode and is a simple way to create short vectors by typing in their values. Try this:

> blastoff = c(5,4,3,2,1)

and look at the created vector blastoff by entering its name at the prompt. A handier way to generate this particular vector is with the integer sequence operator “:”. Try each one of these:

> 5:1

> 1:30

> 3:6

> 10:(-20)

If you would like to both save *and* display an object, enclose the assignment statement in parentheses:

> (blastoff = c(5,4,3,2,1))

Note we can nest functions, arithmetic operators, etc. in expressions; for example:

> seq.length=length(3:6)

First creates the vector resulting from the command 3:6 and then applies the length() function to this vector. The vector 3:6 is not saved. Look at seq.length now.

## 2.2 Factors; coercion

A factor object is superficially similar to a character vector. Let’s look at one:

> Mgroup

These are group labels for the 72 “monarchs” in the data. You see no quotes around each element, though, which is a clue that Mgroup is not a character vector. To confirm this, just ask:

> is.character(Mgroup)

So what the heck is it? Look at its attributes:

> attributes(Mgroup)

Here you should see vectors called **levels** (the three unique character strings that occur repeatedly in the factor) and **class**, which tells you that Mgroup is a factor object. Factor objects are designed for analysis of variance and similar “group-wise” statistical analyses. The “levels” are actually stored as integers 1,2,3…. (try > mode(Mgroup)). Therefore they are very efficiently stored – integer data are cheaper to store than character data. Only the unique character data in $levels needs to be stored, in this case 3 character strings instead of 72.

Most data-input functions such as read.csv will, by default, read any column having non-numeric data as a factor object. Most R functions designed for character data automatically convert factors into character data when needed; however, if you feel strongly that you want to change a factor into a character vector, you can “coerce” it using the as.character() function:

> Mgroup.cvec = as.character(Mgroup)

This transforms the factor Mgroup to a character vector Mgroup.cvec - look at it now and you’ll see the quotes.

There are many coercion functions for forcing objects of one class/mode to be a different class/mode**:** as.numeric(), as.vector(), as.matrix(), as.data.frame(), and so on. Be careful about coercion, though – for example if Mgroups was something that could *not* easily be changed into a character object, trying to coerce it in this way might produce a strange beast, indeed. When this happens, if you are lucky, there will be an (undecipherable) error message with no action taken; if you are unlucky - and if you try to store the new object using the same name as the original - you will destroy the original. When you use coercion, always test the result first to make sure that the created object is what you meant to create.

## 2.3 Matrices

R has a number of built-in data sets we can play with; the simple command > data() will list the names of all these data sets with a brief summary of each. To access any one of these data sets, use the **data** function. Let’s load one of these data sets now, and then coerce the resulting object to be a matrix:

> data(USArrests)

> USArrests = as.matrix(USArrests)

Check your workspace with ls(); see the new object there? Type its name to look at it:

> USArrests

After the “as.matrix” coercion, this object is a numeric matrix with information on incidence rates of certain crimes by state (rates shown are per 100,000 residents; this data was collected some years ago). In general, a matrix in R is a two-dimensional array of values all having the same data mode. The rows are the first dimension; columns are the second dimension. This particular matrix is of dimension 50 x 4; if we were not sure we could ask:

> dim(USArrests)

When you look at this matrix, the words you see on screen are not part of the matrix’ values – they are the row.names and col.names. Every matrix has row and column names - though they might look like numbers (the default).

You can extract individual matrix elements by providing row and column indices in

brackets, e.g. the element in row 3, column 1:

> USArrests[3,1]

is the murder rate in Arizona that year. You can also extract entire rows and columns, e.g. row 3, by leaving the column value blank:

> USArrests[3,]

or all of column 1, by leaving the row value blank:

> USArrests[,1]

or exclude column 4 with a “**-**“ sign:

> USArrrests[,-4]

Note that the results of the above statements are not matrix objects, though; they are vectors. You can also extract elements, or entire rows or columns, using names; e.g.

> USArrests[,”Murder”]

for the first column. More on referencing and extraction of elements later!

Finally, two very useful functions for making matrices are rbind() and cbind(), which bind vectors of the same length into matrices by rows or by columns. Try this one:

> cbind(1:10, 10:1)

## 2.4 Data Frames

A data frame in R is a rectangular array with rows and columns like a matrix, but its columns may be of different modes – the data frame can have some numeric columns, some character columns, some factors, etc; the data must be of the same mode within any column. When you read data into R, for example with **read.csv**, the created result is usually a data frame. Some very important statistical functions will operate only on data frames. Let’s create a data frame from our monarch vectors using the data.frame() function:

> monarchs.dfr = data.frame(Mgroup,Mname,yrslived)

This binds the three vectors (which must have the same length) into a data frame. Type the new data frame’s name to look at it:

> monarchs.dfr

Now look at the data frame’s attributes:

> attributes(monarchs.dfr)

Notice that the column (variable) names of a data frame are just called **“**names”, and the row (observation) names are called “row.names**”**. These can be conjured up as working character vectors using the names() or row.names() functions:

> names(monarchs.dfr)

> row.names(monarchs.dfr)

Unfortunately, the names() and row.names() functions don’t work on matrices! More about this later.

It is dangerous to have the same data occurring redundantly in a workspace, so let’s remove the three vectors that we used to create the data frame monarchs.dfr:

> rm(Mname,Mgroup,yrslived)

The individual elements, rows, or columns of a data frame can be referenced or extracted using the square-bracket technique (numbers or names) as was done for matrices. Alternately, columns can be referenced / selected using two-level names connecting with a dollar sign like this: dfrname$colname. Using the column name alone will not work unless the column already exists as a separately named object - or unless the data frame has been “attached”.

Did you notice that there are now no quotes on the values in Mname? What gives? Ask:

> str(monarchs.dfr)

Note: we use str() (for “structure”) as a useful alternative here—note how it returns the type of both the dataframe and all its columns! So what is the form of Mname?

The data.frame() function coerced the character vector to be a factor object when it created monarchs.dfr. To prevent this from occurring, you may use the I() function, which tells R to leave the object “as is”:

> monarchs.dfr = data.frame(Mgroup,I(Mname),yrslived)

You can quickly obtain a summary of all the variables in a data frame, or columns in a matrix:

> summary(monarchs.dfr)

The actual printed information summary provided for each variable depends on whether the variable is numeric, character, a factor, etc. The function summary() can be used to summarize a wide variety of different objects, including results of complicated data analyses. To review the first few records in a data frame, the command head() can be handy.

## 2.5 Lists

Lists are like glued-together strings of objects of possibly very different structures, lengths, and modes, collectively given one name. We have already seen some lists – the attributes of any object are a list, for example. Let’s extract this list from our new data frame monarchs.dfr:

> attlist.monarchs = attributes(monarchs.dfr)

> attlist.monarchs

There are three members in this list, called $names, $row.names, and $class. In this case they are vectors of varying modes and lengths. Individual members of any list can be accessed/extracted using multiple level names like listname$membername. Alternately, the first member of a list can be accessed / extracted using double square brackets: listname[[1]], the second member by listname[[2]], and so on. Beginning users in R have a hard time distinguishing when to use single brackets and double brackets--be careful to use the correct form!

The members of a list can themselves be lists. For one of these nested lists you might use a double-dollar-sign like this:

> list1$list2$myvec

You might use this if you wanted to extract the vector myvec, a member of the list list2, which in turn is a member of the list list1. Oy!

Many computational and statistical functions in R create and return a list as their output – after doing complicated calculations resulting in a number of different potentially interesting numerical results, the function sticks them all together in a single list and returns that list. For example, consider the eigen() function for matrices: if A is a symmetric nonnegative definite matrix,

> elistA = eigen(A)

is a list with members elistA$values (a vector of eigenvalues of A) and elistA$vectors (a matrix of eigenvectors of A).

## 2.6 Functions

Most work in R is accomplished by functions, programs which are analogous to subroutines in Fortran or C. To fully understand what a built-in R function does, you may need to read (and reread, and reread) its help file (see below), and probably experiment extensively with the function.

Typically, you pass a function some objects (its arguments), and from these it creates and returns one (1) new object (which may be a list of several created quantities). There are exceptions, though – for example, the q() function to quit R usually receives no user-specified arguments, and many plotting functions create graphics but do not return any output. If an object is returned from a function, it will not be saved unless it is assigned a name in the function call. Any intermediate calculations the function performs are not saved after the function terminates; this is a good thing, as it prevents clutter in your workspace.

A full, unabbreviated function call, with the output assigned the name new.object , usually looks somewhat like this:

**>** new.object = **fctname(arg1**=object1, **arg2**=object2,…**)**

The objects passed as function arguments might include numeric or character constants, names of objects in the workspace, or evaluable expressions. Some or all of the arguments may be optional – optional argument values, if unspecified, will have defaults supplied by the function. These default values may be expressions involving other arguments.

As an example, consider the function seq(), which generates evenly spaced sequences of numeric values. It has no required arguments, but calling seq()alone generates a sequence vector from 1 to 1 by 1--not very useful. The most important arguments to seq()are from, to, by,andlength. The arguments **from** and **to** specify the endpoints of the sequence (which will be a decreasing sequence if from’s value is larger than to’s). Usually, either the argument by, which specifies the incremental spacing in the sequence values, or the argument length, which specifies the length of the sequence vector, is also specified - but not both, since the value of by will determine length and vice versa. Each of the following statements will generate (but not save) a vector (-2.0, -1.5, -1.0,…,1.5, 2.0). Try them:

> seq(from=-2, to=2, by=0.5)

> seq(from=-2, to=2, length=9)

Abbreviated function calls omit most of the “arg=” junk and just list the argument values in order:

> seq(-2,2,0.5)

> seq(-2,2,length=9)

Notice that in the second call we had to say length=9 because length is not the third argument in the argument list for seq. Try that last command again without the word “length=”. Can you see the errors waiting to happen in a hasty function call?

As was previously mentioned, functions are sometimes generic, which means that they do different work depending on what kind of objects are passed to them. A simple example is the diagfunction in linear algebra. If A is a matrix, diag(A)returns a vector which is the main diagonal of A(the values in the matrix which have row number equal to their column number). If A is a vector, diag(A)returns a diagonal matrix: a square matrix with the vector A’s values on the main diagonal and 0’s elsewhere.

## 2.7 Other Object Types

Also worth mentioning are so-called “ordered” objects, which are like factor objects except that the levels of the factor are naturally ordered (like Freshman, Sophomore, Junior etc). There are also time series objects, which are like numeric vectors but with some special attributes which are helpful for time series analyses. Other important types of objects include arrays, which are generalizations of matrices to more than 2 dimensions: a 3-dimansional array is a “cube” of data values. Arrays are very useful for contingency table analyses of more than two classifying variables. There are many other object types!

# 3. MORE ABOUT GETTING HELP

Perhaps the most important skill for you to develop early on is reading help files. If you know the exact spelling of the function name or topic which you are curious about, enter the term in the far-right dialog box (with magnifying glass icon) under the Help tab of the lower-right frame or use the dialog box.

> help(topic)

Or just

> ?topic

Here, topic is usually a function name, though entering a built-in data set’s name gives you background information (“metadata”) on the data set. Let’s look up information about the function **rnorm**, which generates normally distributed pseudo-random variables:

> help(rnorm)

Each function help file starts with a general Description of what the function does, followed by lines showing the function’s arguments. In this case, we see

> rnorm(n, mean=0, sd=1)

Arguments which are required are listed first, with no “=” signs; rnorm will generate a vector of length n whose elements are realizations of Normally distributed pseudo-random variables with expected value specified by the argument “mean” and standard deviation by the argument “sd”. We must specify the length of the desired vector, n, or the function terminates with an error message. The other arguments are optional; if we do not specify values for the mean and/or standard deviation they are taken to be 0 and 1 respectively by default. Try it: the first command below generates a vector of 60 pseudo-Normal random variable values with mean 100 and standard deviation 15. The third command constructs a Normal quantile plot of these values. If the values indeed come from a Normal distribution, the plotted points should approximate a straight line.

> normvec = rnorm(60,100,15)

> normvec

> qqnorm(normvec)

The help file section titled Value explains what sort of object (if any) is returned by the function - in this case a numeric vector. There may also be a Details section that gives some explanation as to how the function accomplishes its work; there may also be References. One of the most important parts of the help file is the See Also section, which provides cross-references to other functions which are related to the one whose help file you’re reading. Often the function you’re reading about may not do exactly what you want, but one of the cross-referenced functions will. At the very bottom of the help file there are usually some Examples, but these are rarely as helpful as they could be.

Sometimes you do not need to see all the help file for a function, but only need to be reminded of its argument names and/or order. The function args is ideal for this reminder…try

> args(rnorm)

Certain important help files are hard to find because they have non-intuitive keywords. For example: help(Logic), help(Syntax), help(Arithmetic), help(Comparison)**;** most people wouldn’t guess to capitalize the first letter for these important help topics, but R is case sensitive, so if you type help(logic)**,** R will reply that there is no documentation for this topic. In addition, some words, particularly operators, need to be enclosed in quotes to obtain help ( e.g. help(“if”), help(“%\*%”)).

If you do not remember the precise name of the function or topic for which you want help, there is an interactive search facility; invoke it with

> help.start()

You can also use help.search(keyword)or simply??keyword. This command will generate a listing of R functions and libraries containing the keyword; perhaps the function you’re looking for will be in this list.

Similarly,

>apropos(“logic”)

will provide a list of all R commands that contain the string “logic”. Note that apropros() suffers from the same shortcoming as a regular help() search—it’s case sensitive.

Finally, if you want to know *exactly* what a function does, just enter its name without parentheses at the prompt and R will (sometimes!) obediently display the object itself: the function code, line for line, on the screen. If you are proficient in R, this might occasionally be helpful…but if not - good luck understanding the result! Try this for the function ls;

> ls

# 4. MANAGING YOUR OBJECTS

As already seen, we can remove objects from the workspace using the rm() function. Enter ls() first and then clean up the junk, saving only monarchs.dfr:

> rm(attlist.monarchs, blastoff, Mgroup.cvec, my.prez, normvec, seq.length, USArrests)

Now use ls()again to verify that only monarchs.dfr remains in the workspace. General advice: use rm() often to keep your workspace free of old objects which you will never use again. Note that rm()removes objects one-by-one or all at once (see Section 1) and does not use “wildcards”; because of this, cleaning up a messy workspace can be tedious.

The objects you work with, whether you create them or not, reside in a number of stored directories; when you type an object name in a command, R tries to find it by searching a number of directories in a given order, called the “search path”. To see the current search path directory names, type

> search()

The first directory listed, .GlobalEnv, is what R calls your current workspace. Any object which you create will be stored there if you assign it a name. The other directories in the search path are read-only (cannot be modified); in the list shown here they are all built-in directories containing R functions, data sets, and so on.

As already seen, use the ls()function to list the names of the objects in your workspace. If you see the word character(0), this means your workspace is currently empty; this should be the case if you just started up R for this section and did not save your workspace from a previous session. You can see a higher level directory’s contents by supplying a number, e.g.

> ls(9)

will display the object names of the 9th directory of the search path. You may also refer to any directory in the search path by its name, e.g. ls(“package:base”)

You can add certain kinds of objects to your search path for the current interactive session. For example,

> attach(monarchs.dfr)

attaches the data frame monarchs.dfr in position 2 of the search path, which (like all positions except the workspace) is read-only. You can also attach lists or previously created Rdata directories (using a full file address specification, in quotes) which may contain data objects and/or functions you created in earlier work sessions; we’ll do that later.

Check the search list again:

> search()

And look at the objects you loaded into position 2:

> ls(2)

Now that monarchs.dfr is attached, you can access its columns (variables) without clunky two-level $-names, like this:

> hist(yrslived)

When you want to remove monarchs.dfr from the search list, type

> detach(monarchs.dfr)

or

> detach(2)

or, simply quit R.

Save your cleaned-up workspace now. If you want to use this saved workspace in a future session, there are several ways to load it. The easiest way is to launch R directly at that location by double-clicking the R logo next to the file you want to use as a workspace. If instead you choose to launch R in some other way, you can add a workspace after-the-fact: go to File, then Open File, and then find the workspace you want to load. A third way is to load the previously-created workspace into search-path position 1 with the attach() function; for example, this command might look something like this

> attach(“C://My Documents//classes

//stat540//testing123.RData”,pos=1)

There is danger here, though - this replaces the current workspace with the specified one. Note that only .RData files can be loaded in position 1 (no data frames or lists).

If you want to use objects in a previously created R workspace as read-only objects for a current session (for example, if they are functions that you want to use but not change), use a command like the above with pos=2 (or with no position specified), and the workspace will be attached as read-only in position 2.

It is usually a bad idea to keep the same function or data object in more than one directory. If you change one of the copies, you may forget to change the other, and at some later date you will be referring to the object and not know which of the two you’re working with. It is also a bad idea to have two functions of the same name in the search path; when you call the function, R will use the first function of that name it finds in the search path. Likewise, if you use a name for a data-object that is also being used for a built-in function, e.g. “plot”, whenever you call the function plot you may see a message like this one:

Looking for object “plot” of mode function. Ignored one of mode numeric…

When you quit an R session, R will prompt you to save the workspace, even if you have just saved it as a named workspace. You should answer “No” if you have just saved the workspace under a full name; otherwise, an additional workspace labeled “.Rdata” will be saved as well.

# 5. GETTING DATA INTO R

## 5.1 Creating Data

First, there are a number of handy functions for creating vectors and matrices in R. We have already seen the c()and seq()functions, and the integer sequence “:” infix operator. Also useful is the rep() function, which creates a vector by repeating a single value or vector a specified number of times….try these:

> rep(1,10)

> rep(1:10,10)

> rep(“Gamecocks Rule”,1000)

We can easily generate vectors of pseudo-random numbers in R; we have already introduced the rnorm()function for generating Normal random variates; there are many others, including runif(), rchisq(), rbin(), etc.; enter

> help.search(“distribution”)

to see a partial list.

Another useful data-creating function is the matrix()function. The command

> constmat = matrix(value,r,c)

which creates and stores a matrix called constmat with *r* rows and *c* columns (these must be whole numbers) with every entry equal to value (which may be of any mode – numeric, character, logical etc). Examples:

> zeromat = matrix(0,5,3)

> onesvec.col = matrix(1,10,1)

Note the difference between rep(1,10) and matrix(1,10,1); the former is a vector of length 10 – it has no orientation. The latter is a 10x1 matrix – a column vector in the matrix algebra sense.

The matrix function can also reshape vectors into matrices. For example, suppose bigvec is a vector of length 200, and we would like to reshape it into a 100 row, 2 column matrix. An important question: should the vector’s elements be written into the matrix a row at a time, or a column at a time? The default is to write by columns, but if the desire is to write by rows, add the option byrow=T:

> bigvec = 1:200

> mat.bycols = matrix(bigvec,100,2)

> mat.bycols

> mat.byrows = matrix(bigvec,100,2,byrow=T)

> mat.byrows

I never remember if the default is to write the matrix by rows or by columns, so I always just specify byrow=Torbyrow=F**.**

## 5.2 The read.table()function and friends

More often than not, a data analyst will need to import data which is in roughly columnar form into R as a text file, perhaps created by some other software. If each data record consists of the same number of items, on a single line, the function read.table()for creating a data frame from the text file was long recommended. This section will discuss read.table() in detail, but read.delim(),which has similar features, is often more robust and should be used first when handling tab-delimited or comma-delimited (see read.csv()as well) data sets. Note that we will need missing values in the text file to be represented by some unambiguous missing value code, and values on each line separated by some well-defined delimiter (usually blanks, tabs, or commas). Items need not be in exactly the same columns on different lines. It is recommended that you read (and reread) the help file for read.table() carefully, to find out exactly what its capabilities and limitations are.

To use read.table()most effectively, let the first item of each record be a character variable to be used for row names; any names that have embedded blanks should be enclosed in quotes. Then, add a row at the top of the text file with the column names you desire, but no name for the row names column. If each record’s data values are separated by white space or tabs, and missing values are represented by the R missing value code NA (which stands for “not available”), then the command

> my.dfr = read.table(“filespec”)

should create a data frame called my.dfr with the desired column and row names. Important note: the file specification filespec should use double *forward* slashes wherever a normal file specification would use single *back* slashes, e.g. E://myfile.txt for a text file myfile stored on a thumb drive. If the file specified by filespec is not in the directory where the current workspace resides, then the specification must include the full directory path to the file. This problem can be avoided entirely by changing the default directory; select *S  
ession* then *Set Working Directory* then *Choose Directory…* from the R toolbar, and then choose the directory that contains, e.g., myfile.txt. The usual default directory would be a subdirectory with the set of R program files—it is generally convenient to change this directory to your personal workspace at the start of each R session.

For example, suppose the text file below, brainbod.txt , which lists typical body weights (kg) and brain weights (g) for 15 terrestrial mammals, exists in the same folder where R was opened. Note there are three items in every row, the first being a unique name for the row. Since the first record has one less item, these are presumed to be the **names** for the data frame to be created. Note the missing value in row 6, column 2.

|  |  |  |
| --- | --- | --- |
|  | bodywt | brainwt |
| afele | 6654.00 | 5712.00 |
| cow | 465.00 | 423.00 |
| donkey | 187.00 | 419.00 |
| man | 62.00 | 1320.00 |
| graywolf | 36.33 | 119.50 |
| redfox | 4.24 | NA |
| narmadillo | 3.50 | 10.80 |
| echidna | 3.00 | 25.00 |
| phalanger | 1.62 | 11.40 |
| guineapig | 1.04 | 5.50 |
| eurhedghog | 0.79 | 3.50 |
| chinchilla | 0.43 | 64.00 |
| ghamster | 0.12 | 1.00 |
| snmole | 0.06 | 1.00 |
| lbbat | 0.01 | 0.25 |

The command

> brainbod = read.table(“brainbod.txt”)

will create a data frame with two columns, names bodywt and brainwt, and row.names present. Note that if there had been any character data other than the row names in the text file, their columns would have been converted into factor objects. To preserve character columns as character objects, add the option as.is=T to the read.table() call. It is a highly recommended exercise to copy the above table, paste it into a text file, and try to read it into R as a data frame as described above.

Usually, missing values in a raw text file are not represented by the R missing value code NA. If this is the case, there is an option na.strings in read.table that allows specification of a different missing value character, e.g. the naked decimal “ .” in SAS: na.strings=” .”. Note there is a blank before the decimal in this specification, to make sure that decimals imbedded in numbers are not interpreted as missing values – this specification assumes that numeric values less than 1 are represented in the text file using leading zeros, i.e. “0.XXX”. It may be obvious from this that one must BE VERY CAREFUL in reading large, complex data sets (in any language, actually). When there is a great deal of data, anything that can go wrong, will go wrong. There will probably be “typos”, for example. Always check the results very carefully, and be patient.

Suppose the first entry of every row of the text file is a numeric value, not to be used as a row name. If, as above, column names have been typed as the first row of the file (and now every record has the same number of items as there are column names), try

> my.dfr = read.table(“filespec”, header=T)

This will generate row numbers as default row names. Or, you can add the option row.names=charvec.row in the call to read.table(), where charvec.row is a workspace character vector you would like to attach to the data frame as row names. Of course, you can also change row names easily after the fact with a command like

> row.names(my.dfr) = charvec.row

If for some reason the (column) names of the data frame cannot be included as the first row of the source text file, default **names** V1, V2,etc. will be generated; to avoid this, you can use the option col.names=charvec.col in the call to read.table(). Or, change them after the fact with something like

> names(my.dfr) = charvec.col

As an alternative to reading text files, if you wish to read an Excel workbook, you may download and install R’s xlsx package, and use read.xlsx() to input a given worksheet.

## 5.3 The scan() function

In rare cases of very complex data to be read into R, the read.csv() and related functions may not have the needed flexibility. In these instances, the scan() function, more flexible but more difficult to use, may fill the bill. The help file is difficult to read, but worth the effort if you will be inputting messy data. Useful optional arguments to scan() are the what, multi.line,andwidthsarguments. For example, suppose we must read a data set with several thousand records, where each record has 50 values, some numeric, some character, in variable-width fields, and each record is written in several lines. After a careful search-and-change in the text file to change missing values in the data to NA, the following generic call should work:

> my.dfr = as.data.frame(scan(“filespec”,

+ what=list(name1=0,name2=””,name3=0,……,name50=””),

+ multi.line=T))

Here, the “**+**” signs are supplied by R as a continuation prompt whenever you type a statement which reaches beyond the end of one line. The what argument in this case supplies a model for a single dummy record of the text file, with variable names: name1=0 means the first value in each record of the file is numeric; name2=”” means the second value is character, and so on. Since the object assigned to what is (in this case) a list, scan() creates a list from the text file with supplied element names name1, name2, and so on. The as.data.frame function converts this list into a data frame; the list element names will become the names for the data frame my.dfr. The multi.line=T option causes **scan** to ignore line breaks in the text file.

If the data in the text file are in fixed-column format, in which case there are often no missing values represented in the data, the widths argument would probably be needed. To use it, include widths=c(3,4,2,…) anywhere in the scan argument list, where the values inside the **c**() are the field widths for each variable in the data record. You may also want to include the option strip.white=T so that character variables’ values will not include leading or trailing blanks.

Interestingly, scan() is also taught as the simplest method for entering data, as it can be used to read a single column from keyboard input or a text file as an R vector object.

# 6. GETTING RESULTS OUT OF R

Any results or commands written in the R command window (the “console”) can be selected, copied and pasted into a word processor and then edited.

The easy way to get an entire data frame out of R for use by other software or spreadsheet program is to export it as tab-delimited .txt file or comma-delimited .csv file. If your R data frame is called newdata.dfr, the following command should create a .csv file newdata.csv in the current directory.

> write.csv(newdata.dfr, “newdata.csv”)

To write the .csv file somewhere else, use a full file specification in the quotes, with double backslashes where one would usually have single slashes.

If you do not want to create a .csv file, a common situation is the need to output a matrix to a text file. R has implemented write.table() to do this. It saves the matrix row and column names by default; if this is not desired, we will likely want to use the following form to write to the current directory:

> write.table(mymat,”mymat.txt”,row.names=F,col.names=F)

We will discuss writing our own functions soon. For backup and sharing these with others, we will probably want a text file representation of the function. This can be accomplished easily using the fix() function:

> fix(myfct)

This opens a text editor (Notepad in Windows) to allow one to modify the function code. When finished, go to File ⇒ Save As and type the desired name for the text representation. Then go back to File and Exit.

If you would like to export several objects and/or functions without taking the time to deal with them individually, the **dump** function will get everything out quickly, though you may not like what you get:

> dump(c(“object1”,”object2”,…),”miscstuff.txt”)

Often, especially in debugging a function, you will want to print messages to the screen or to an output file. The print() function will print any character or numeric object to the screen, or to a text file if a file specification is given. To combine character and numeric objects into one message, use paste() to glue them together into a character string:

> print(paste(“Now starting iteration number”,iter))

Here, iter is an integer-mode counter inside a loop; paste() automatically converts its current numeric value to character and adds it to the first character string. You can use paste()in many settings; for example to title graphs using current values of variables being processed.

Other functions designed to get results out of R include dput(), sink(), pdf(),and postscript()for plots and other graphics. Look at help files for these, if interested.

# 7. ARITHMETIC

For all details, reading the help(Arithmetic) and help(Syntax) files is recommended, along with experimentation with the operators discussed here. The infix operators for addition (+), subtraction (-), multiplication (\*), division (/), exponentiation (^), and modulo arithmetic on integer objects (%%) are most common. Operator precedence is standard and is detailed in the Syntax help file; use parentheses liberally to be safe. The general use of these infix operators is

> new.object = object1 op object2

where op is an arithmetic infix operator and object1, object2are integer, double-precision, or complex(-number) objects such as vectors, matrices, or arrays. If object1 and object2have exactly the same “size” (length or dimensions), the created result stored in new.object is of that size (has the same length / dimensions), and is produced by elementwise application of the operator to the two objects. If object1, object2are *not* of the same length/dimensions, the smaller one is “recycled as necessary”, with a warning if it is only fractionally recycled. This can be a nice feature, but can lead to major errors without error messages if one is not careful. For example, in the expression

> new.object = (Mymat ^ object2)

if Mymat is a matrix and object2is a scalar (a single value…a vector of length 1), then new.object is a matrix of the same dimensions as Mymat obtained by exponentiating all of its values to the power given by object2. If object2 is a vector of length 2, it is not really clear what “recycling as necessary” will give you…and there may be no error message….so check the result carefully.

The t()function transposes a matrix (exchanges rows and columns); it has no effect on an R vector because vectors in R have no orientation. The command

> colvec = as.matrix(myvec)

coerces the R vector myvec to be a column vector (an nx1 matrix, where n = length(myvec) ). The matrix and diag functions for creating matrices have already been mentioned; if the argument to diag() is a single positive integer, an identity matrix of that size is created. Try the command

> diag(10)

If Aand B are matrices the matrix product AB defined in linear algebra texts requires that the number of columns of Aequals the number of rows of B. This product is obtained using the matrix-multiplication infix operator %\*%, i.e. by a command like

> C = A%\*%B

Note that if Ais a matrix, A^2 and A\*A are the same, obtained by elementwise squaring or multiplication, but A%\*%A is very different (and might not even be defined).

If b is a vector such that the system of equations Ax=b is well defined and has at least one solution x, then solve(A,b)will provide a solution. If B is a matrix, solve(A,B)will be a matrix of the same dimension as B obtained by separately solving the several systems corresponding to each column of B. The inverse of a nonsingular matrix is obtained by

> Ainv = solve(A)

Note though that if you want to solve a system of equations Ax=b it is numerically more accurate and faster to use x = solve(A,b)than to create Ainv and then multiply to obtain x = Ainv%\*%b.

The functions nrow(), ncol(), and dim()produce simple numeric vectors describing the size of a matrix supplied as their argument. The useful functions rbind()and cbind()to create matrices by pasting together vectors of the same length and mode have already been mentioned.

Create a matrix A:

> A = c(4,1,3,7,-1,0,2,2,6,5,-2,1)

> A = matrix(A,3,4,byrow=T)

Create a vector a:

> a = c(1,6,3,2,1)

# Since vectors in *R* do not have orientation, a as defined above is neither a row vector nor a column vector. When used in matrix manipulations, *R* will try to orient vectors as needed to carry out the calculations. Since *R*’s judgment in orienting vectors is not foolproof, it may be necessary at times to force a row- or column-orientation for *R* “vectors”. This can be done by shaping them into a 1-row or a 1-column matrix. For example, the following command forces a to be a column matrix:

> a = as.matrix(a)

Or, we can force ato be a row vector (a 1x5 matrix):

> a = matrix(a,1,5)

Now create two matrices and matrix-multiply

> Mat1 = c(1,2,1,-1,3,0)

> Mat1 = matrix(Mat1,2,3,byrow=T)

> Mat2 = c(1,2,3,4,0,-1,-2,1,3)

> Mat2 = matrix(Mat2,3,3,byrow=T)

> Matprod21 = Mat2%\*%Mat1

Oops! ...that was the wrong order – that product is not defined since the number of columns of Mat2 is not equal to the number of rows of Mat1. Just wanted to see if you were paying attention…here is what was meant:

> Matprod12 = Mat1%\*%Mat2

Now create a matrix M and invert it:

> M = c(3,4,5,1,2,6,7,1,9)

> M = matrix(M,3,3,byrow=T)

> Minv = solve(M)

Check that MM-1 and M-1M are identity matrices , except for round off error (matrix inversion is a tricky business, numerically speaking):

> M%\*%Minv

> Minv%\*%M

> round(M%\*%Minv,10)

Other functions relevant to matrix arithmetic will be introduced if needed; some important ones are eigen, qr, chol, kronecker**,** andsvd**.** Most matrix algebra operations, expressions, etc. work with purely numeric data frames as well. If not, first try (carefully) coercing the data frame to be a matrix using as.matrix().

# 8. LOGICAL OBJECTS AND CONDITONAL EXECUTION

An object of mode logical is a vector, matrix, array etc. whose elements consist entirely of TRUE or FALSE values (sometimes abbreviated just as T and F). To see one, load a built-in data frame on autos from Motor Trend magazine (a longgggg time ago…)

> data(mtcars)

and try

> guzzlers = (mtcars$mpg < 20)

> names(guzzlers) = row.names(mtcars)

> as.matrix(guzzlers)

The first command of these three creates the logical vector guzzlers, with length equal to the length of mtcars$mpg, whose value is TRUE if the corresponding car in mtcarsgets less than 20 miles per gallon of fuel, and FALSE otherwise. The other commands attach names to the vector guzzlers and display it as a column matrix on the screen.

Logical objects are invaluable for subsetting data (Ch. 9) and for conditional execution of statements using if() clauses. Most logical objects are generated with a statement like

> logical.obj = object1 comparison.op object2

where object1 and object2 are usually numeric constants, vectors, matrices, or arrays (though character objects can be used for alphabetical comparisons) and **comparison.op** is usually one of the following logical infix operators:

> < >= <= == !=

The latter two operators stand for “equal” and “not equal”, respectively; note that the single equal sign “**=**” is not a comparison operator. Be sure to leave a space between “<” and a negative sign in doing comparisons, or use parentheses to separate these.

As in arithmetic expressions, if object1 and object2 in the statement above are the same length / dimensions, the comparison is done elementwise and the created logical object is of the same length / dimensions. If object1 and object2 are not the same length / dimensions, the smaller object is “recycled as necessary”, generating a logical object whose length / dimensions match the larger of object1 and object2. In particular, if object2 (e.g.) is a scalar, as in the case of guzzlers above, the command compares every element of object1 with that scalar and returns a collection of TRUE/FALSE results with the same size/dimensions as object1.

Logical objects can be combined in a number of ways. If logical.obj is a logical object, then !logical.obj is its complement, i.e. an object of the same size and structure as logical.obj with TRUE and FALSE reversed in each position. If logical.obj1 and logical.obj2 are two logical objects, then

> logical.obj1 & logical.obj2

(& is the “and” operator) evaluates to TRUE in every component where the compared elements of logical.obj1 and logical.obj2 are *both* TRUE, and evaluates to FALSE otherwise. If these objects are not of the same length / dimensions, the smaller one is recycled as necessary. In contrast,

> logical.obj1 | logical.obj2

(| is the “or” operator) evaluates to TRUE in every component where *at least one* of the compared elements of logical.obj1 and logical.obj2 is TRUE. There are also “sequential-and” and “sequential-or” operators &&and ||which have similar results but can be more efficient – see help(Logic).

There are several important functions which use and/or return logical arguments. We have already seen the query functions is.matrix, is.character, etc., which return a scalar TRUE or FALSE. Other important functions involving logical objects include all(), any(), all.equal()and identical()**.** For example,

> all(logical.obj)

evaluates to a scalar TRUE if and only if *every* value in logical.obj is TRUE.

Similarly,

> any(logical.obj)

evaluates to a scalar TRUE if and only if *at least one* value in logical.obj is TRUE. If you would like comparisons involving missing values to be ignored, add the argument na.rm=Tinside the parentheses – more about missing values later.

The function identical() compares any two R objects and returns a scalar TRUE if and only if they are identical in every respect - including structure and mode. For example,

> identical(1., as.integer(1))

is FALSE because the second object is not a double precision numeric value; identical() will also signal false when comparing a matrix and a data frame, even if their dimensions and values are identical.

The operator ==and the function identical() may signal FALSE when comparing numeric objects which are equal up to roundoff error (i.e. equal up to machine precision); this can be undesirable. The function all.equal() compares two objects testing for “near equality”, and the tolerance for judging equality of numeric objects can be adjusted (see the help file). If it finds no differences using this this tolerance, all.equal() returns TRUE; otherwise, it returns a character vector describing the differences found.

One can use logical expressions in an if() clause to place a condition on execution of a statement or a group of statements. A generic if() clause might look like this:

> if(logical.condition) {

statement1

statement2

}

else {

statement3

statement4

}

Here, if logical.condition is TRUE, the statements in the first group of brackets are executed; otherwise, the statements in the second group of brackets are executed. If there is only one statement in a group, the brackets are not necessary. The elseclause is not required if action is only to be taken when logical.condition is TRUE. When using the else if/else clause, always make sure the else if/else statement appears on the same line as the right-hand brace (as shown above). Otherwise, R completes execution of the if statement, and can’t interpret else correctly.

The logical.condition in the above clause should evaluate to a simple (scalar) TRUE or FALSE; if it evaluates to a logical vector of length 2 or more, or a matrix, *only the first element will be checked to decide if the condition holds* (mercifully, with a warning message). If you would like to execute an if() clause only if *all* elements of the numeric vector testvec are zero (within machine epsilon), here are several *bad* ways to do it:

> if(test.vec == 0) …(condition evaluates to a vector)

> if(all(test.vec == 0)) …(tiny roundoff error causes FALSE )

> if(all.equal(test.vec,0)) (all.equalcan return char.vec.)

The recommended technique for this sort of conditional execution is to nest all.equalwith identical:

> if(identical(all.equal(test.vec,0), TRUE)) …

The ifelse() function carries out mass-production conditional execution over a vector or matrix or array, making it one of the most commonly used functions - absolutely invaluable to avoid messy and time-consuming loops (discussed in Ch. 10). A typical call to ifelse() looks like this:

> new.object = ifelse(logical.obj,expression1,expression2)

where logical.obj can be a vector or matrix or array of TRUE’s and FALSE’s, or an expression that evaluates to one. The created new.object is an object of the same length / dimension as logical.obj with the value of expression1 found wherever

logical.obj is TRUE; otherwise the result of expression2. If the expressions do not generate objects of the same size as logical.obj, they are recycled as necessary.

Here is a useful example of ifelse(), a command to replace missing values in a numeric object my.object with zeros:

> my.object = ifelse(is.na(my.object),0,my.object)

Here is another example which creates a “truncated” mpg vector, replacing mpg values greater than 20 with the value 20:

> mpg.truncated =ifelse(mtcars$mpg>20,20,mtcars$mpg)

Logical objects can also be used in arithmetic expressions! The numeric value 1 replaces TRUE and 0 replaces FALSE. For all details on logical objects and operators, reading the help(Syntax) and help(Logic) files is highly recommended.

# 9. SUBSETTING, SORTING, AND SO ON…

Chapter 2 on objects introduced a few simple ways to subset vectors and matrices (and data frames, and arrays…) using brackets [ ]with names or index numbers inserted to extract a vector element or a row or column from a matrix. This technique can be generalized considerably. Suppose we want to select some of the elements of an existing vector oldvec, and suppose index.vec is a vector giving the indices (numeric positions) of desired elements to extract. The following statement

> newvec = oldvec[index.vec]

Will extract and store the desired elements in newvec *in the order that they occur* in index.vec. For example, load the state data set again:

> data(state)

and extract the last five states in reverse alphabetical order:

> tryit = state.name[50:46]

> tryit

A minus sign on an index indicates that this element is *not* to be extracted from the vector. For example,

> state.name[-(31:40)]

is all the states *except* those numbered 31 through 40 (so, e.g. South Carolina, number 40, would not be included in the extracted vector).

These indexing tricks can be used to easily drop rows or columns, or extract submatrices from larger matrices, e.g.

> newmat = bigmat[-1,]

is everything but the first row of bigmat,

> newmat = bigmat[,2:10]

is columns 2 through ten of bigmat, and

> newmat = bigmat[1:3,1:3]

is the upper-left 3x3 submatrix of bigmat.

We can also use names to extract certain elements of named vectors, matrices, and data frames, though this is less handy when we want to extract more than one index. For example, to study data for only the Carolinas and Georgia, we might subset the matrix state.x77 selecting three rows as follows:

> tristate.region = state.x77[c(“South Carolina”,

”Georgia”, ”North Carolina”),]

> tristate.region

Though cumbersome, using names instead of index numbers can be a safer way to extract values when there are missing values in the data. For example, to do certain calculations, missing values will need to be removed, but after doing this the remaining data items may not be numbered in the way you expect.

A third and very important way to extract elements, rows, or columns is through the use of logical vectors. The command

**>** newvec = oldvec**[**logical.vec**]**

will extract the elements of oldvec only in components for which logical.vec is TRUE, in the order in which TRUE’s are encountered. Note logical.vec must be the same length as oldvec. For example,

> newvec = oldvec[oldvec>0]

extracts the positive elements of oldvec in the order they appear and stores them in newvec. Let’s try to extract the rows of state.x77 corresponding to the defined level SOUTH in state.region:

> south.region = state.x77[state.region==”South”,]

> south.region

Hmmm…is Delaware really part of the South? Notice that when you print the object state.region to the screen, the levels South, North Central, etc. do not appear in quotes (state.region is not a character vector – it’s a factor), but the above logical statement does not work properly without quotes.

The output object does not retain information about the original location of its elements, but these can be recovered by the logical vector itself (e.g., oldvec>0), or use of the which() command, which returns the original indices of the output object. The commands which(x>0) and x>0 store the same information, though in different forms; both forms have their uses depending upon the application.

A number of functions can be useful in generating the information needed to generate index vectors, or to directly subset vectors and matrices. See for example the help files for match(), split(), sample(), unique(), cut(), table(), category(), tabulate(), grep()**,** and charmatch().

The index-vector extraction technique is actually the basis for complex sorting of vectors, matrices, and data frames via the order() function: let

> ord = order(sortvec)

Then ord is an index vector which maps the elements of sortvec to their locations in a sorted version (i.e., ord is a permutation vector). This is easier to understand with an example:

> ord = order(state.area)

> ord

[1] 39 8 7 11 30 21 29 45 20 48 40 19 14 17 46 35 …

which means that the smallest state in area is the state in position 39 in state.area, the second smallest state is the state in position 8, and so on. Since these are indices, one can use the created vector ord with subsetting to create sorted versions of objects. Try this:

> state.name[ord]

and confirm that Rhode Island is indeed the smallest state in area. To list the objects in reverse order of area, try

> ord = rev(order(state.area))

> state.name[ord]

and discover that it is indeed true that California is the third largest state behind Alaska and Texas. Instead of using the rev()function, we could have done this:

> ord = order(-state.area)

We can use the order()function to sort an entire matrix or data frame by the elements of a given column or by the elements of another vector of the same length as a column. To sort the rows of state.x77in order of decreasing population, try this:

> Population = state.x77[,”Population”]

> ord = rev(order(Population))

> state.x77.popsorted = state.x77[ord,]

> state.x77.popsorted

Note however that these statements do not sort other relevant objects in the workspace like state.abbor state.region **,** etc.,so if you later use these in conjunction with state.x77.popsortedthere will be many mismatches, with no error messages.

The function order()accepts any number of vector arguments of the same length, of possibly different modes, as long as each vector can be ordered by its values. If several vectors are passed to order(),the output index vector is the permutation required to perform a nested sort. For example,

> ord = order(as.character(state.region),

(-Population))

is the permutation vector required to sort states first by region (alphabetically), and then within each region by decreasing Population. Try it:

> state.dfr =

data.frame(state.name,state.region,state.x77)

> ord = order(as.character(state.region),

(-Population))

> state.dfr=state.dfr[ord,]

> state.dfr

Note we coerced the factor object state.region to be character here using the as.character() function, since state.region has an implied order by levels but this ordering might not be alphabetical.

There is also a sort()function that accepts only a single vector argument, returning the sorted version. It does not seem to have the same breadth of application as the order()function.

# 10. ITERATION

Recall that R was built from the AT&T package S, which is in the public domain. The commercial package Splus, also built from S, existed long before R; perhaps its greatest weakness (besides its cost) is that it does not loop efficiently. R is reputed to be more efficient than Splus for looping, but nonetheless one should try whenever possible to avoid looping in R. This is often possible since R has many built-in “vectorized” iterative capabilities.

Avoiding loops will be more a matter of breaking old programming habits than anything else. Many tasks traditionally done in loops can be accomplished in R with a single statement, e.g. squaring every element of a 1000x200 matrix Acan be accomplished by the statement A^2instead of two nested loops! Most of the standard built-in functions operate elementwise on vectors, matrices, or arrays, e.g. sin(A),exp(A)etc. producing objects of like size and structure, applying the function elementwise in a very efficient way. With some thought, one can often determine a combination of functions that will do mass-production calculations without looping. For example, to generate a matrix of random numbers, generate a large vector and shape it into a matrix using the matrix()function. The kronecker()function is another useful mass-production function for matrix operations. Be creative – it can save a lot of computer time.

If you MUST loop, try to keep the loop size small, and try to avoid nested loops. The syntax for looping uses either a for or while clause, usually something like this, e.g. to carry out some calculations over each column of a matrix mymat with 100 columns:

for (j in 1:100) {

Statement 1 for mymat[,j]

Statement 2

etc etc

}

One can also use while(logical.condition)to determine entry to, and exit from, a loop. See help(Logic).

As a concrete example, here is a loop to fill a 1000 x 500 matrix with randomly generated numbers from a home-written function called getRV, which unfortunately only generates one random value at a time:

mymat = matrix(NA,1000,500) #for storing results

for (i in 1:1000) {

for (j in 1:500) {

mymat[i,j] = getRV()

} # end j loop

} # end i loop

The text strings starting with # are comments. The indenting shown above is not required.

R has many built-in functions to help with iterative calculations and operations. For example, suppose we would like to apply a function repetitively to the rows of a matrix. For these instances, the apply() function will usually do this somewhat more efficiently than loops:

> new.obj = apply(array, indices, fctname, …)

Here, the function whose name is given by fctname is to be applied to array

(considering vectors and matrices as one- and two-dimensional arrays) over the indices specified in indices: rows=1, columns=2, etc. If indices is not mentioned, the function is applied elementwise over the entire array. The notation “…” in the argument list above refers to the fact that apply will also accept as arguments any arguments that fctname needs.

Here are some examples: Suppose myfct can operate only on scalars:

> newmat = apply(oldmat,myfct)

applies myfct separately to every element in oldmat; the results are stored in the matrix newmat, which has the same dimensions as oldmat.

A common task is to find the maximum of each row of a matrix; the command

> max(mymat)

will not do this, as it will find the single maximum value of the entire matrix. To create a vector containing the maximum value of each row,

> rowmaxes = apply(mymat,1,max)

Since the **max** function will by default deliver a missing value NA if it finds any missing values in its argument, you may want to add the max argument na.rm=T to the call:

> rowmaxes = apply(mymat,1,max,na.rm=T)

Now the function will produce a vector containing the row-wise maxima in mymat, ignoring missing values.

Another way to find maxima in groups is the “parallel max” function pmax. If vec1 and vec2 are numeric vectors of the same length, then

> pmax(vec1,vec2, na.rm=T)

will produce a vector of that length with the maximum of the corresponding matched values from vec1 and vec2, ignoring missing values. One can have any number of vectors, and pmax works for matrices and arrays, also.

Another frequent task is to apply a function to certain groups in the data, the sort of thing a BY statement in SAS does. The tapply() function can do this:

> newobj = tapply(myvec,groups,fctname, …)

Here, groups is a factor, vector, or list of these, each the same length as myvec, whose values or combinations of values specify subgroups for the elements of myvec. The function fctname is a function that can operate on these subgroups of myvec, and “….” again refers to optional arguments for fctname. An example:

> Income =state.x77[,”Income”]

> medinc.byreg = tapply(Income,state.region,median)

> medinc.byreg

This computes median state income by region.

There are two other apply–like functions, sapply() and lapply(), which can apply functions to each element of a list.

# 11. MORE ABOUT GRAPHICS IN R

One of R’s strongest virtues is its integrated graphical capabilities. This primer will hardly scratch the surface on this topic. For a quick tour, try

> demo(graphics)

> demo(image)

Several graphics devices (window types) are available. Most plotting commands will automatically print in the Plots window in the lower right-hand frame, but if you want to open one directly, the command

> windows()

opens the default style device on a Windows operating system and possibly on others…. on a Mac, the equivalent command is>quartz().

*Note:* This is a feature that behaves oddly in RStudio—after your first windows()/quartz() command, another RStudio icon will appear on your toolbar, but its only content will be the new graphics window and any subsequent windows you create. You will no longer have access to the Plots tab while any of these windows are active, so it is best not to explore this peculiarity of RStudio further at this time.

To start, here is a plot of 100 generated standard Normal values in the order generated:

> zvars = rnorm(100)

> plot(zvars)

The most important options available there are under the Export menu; they include

* Save As Image: this allows the graph to be saved as a graphics file in any of several standard formats.
* Save As PDF: this allows the graph to be saved specifically as a PDF file.
* Copy to the Clipboard: if you want to paste the graph immediately into another document. Caution: if using Metafile format to paste into MS Word, the figure may become distorted if you later save the Word document as pdf. You can also use a postscript() command to save copies of graphs.

R has default font styles, colors, box around the plot, etc, which you can modify. These current “graphical parameter” choices for an active graphics device can be viewed and/or modified with a call to the function par(). Look at the default values:

> par()

As you see, there are a great many of these graphical parameters; to gain complete expertise in using R’s graphics, one will likely read the help file for par()many times. The default choice for these parameters can be changed for the current session by a call to par(), for example (don’t do this now please):

> par(pty=”s”,bty=”l”,lwd=2,pin=c(3,3),las=1)

will force all plots made on the graphics device to be square, and to have an “L” shaped box instead of a box completely around the plot, and to have line widths doubled compared to the default width, and to make plots which are 3”x3” in size, and to have axis labels which are horizontal rather than the default (parallel to the axes). Any given choice of parameters can be saved as a list with a name and recalled later in a new R session. Also, whatever choice is currently in use can usually be overridden using keywords in the creation of any given figure.

## 11.1 Scatterplots

For raw data load the state.dfr data, a built-in data set with descriptive statistics for the 50 states in 1977 (apologies for the age of this data).

> data(state)

> ls()

> statex77.dfr = data.frame(state.x77)

> rm(state.x77)

> names(statex77.dfr)

> attach(statex77.dfr)

Chapter 1 showed how to construct single-variable graphical descriptives like pie charts, barplots, stemplots, histograms, smoothed histograms, ecdf’s and boxplots.

The most important graphical tool for studying the association between two numeric variables is the scatter plot. For a simple scatter plot of two variables, specify the variables in a call to plot() (specify the horizontal “x-axis” variable first). For example,

> plot(Illiteracy,Murder)

We can attempt to make the graph more attractive with some options:

> plot(Illiteracy,Murder,las=1,lwd=2,cex=1.2,pch=19,

+ xlim=c(0.3,3),ylim=c(0,15),xlab="Pct Illiterate in

+ State",ylab="Murder rate per 100,000")

> title(“Murder vs. Illiteracy”)

> text(2,3,”The 50 United States”,cex=1.2,adj=0,col=2)

The graphical parameters las and lwdcontrolling axis-label style and line widths have been mentioned already. The choice pch=19 calls for a solid-circle plotting symbol. The cex argument changes font size (“character expansion”): if less than 1, it shrinks the characters; if greater than 1, it expands them. The default axis labels are overridden by xlab and ylab; the default axis limits are overridden by xlim and ylim**.** To learn more about these and many other graphics options, try par().

The titleand text commands add to the existing plot. There are a great number of other “low level” graphical functions that add to existing plots: see the help files for points(), lines(), symbols(), arrows(), abline(), segments(), mtext(), polygon(), legend(), axis()and others. For many of these add-to-the-plot functions the first two arguments to the function are x,y coordinates (or possibly vectors of coordinates) for the location(s) on the plot where text, points, or whatever are to be added, using the current axis system set up on the active plot. For example, in the text statement above, the values 2,3 specify that the provided text string is to be located at those x,y coordinates; adj=0 means that the string will be left-justified at that spot (adj=0.5, the default, will center text, and adj=1 will right-justify text, at the supplied coordinates). The col=2 argument specifies a color number from the current color scheme. Alternately, you may specify color with a character string, e.g. col=”firebrick”. Mathematical symbols can also be used in text strings; see the help file for function plotmath().

The location of a legend may be specified with a keyword from the list "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right"or"center”(example to come). The locations of add-on text or symbols can also be determined interactively with the locator() function. For example, enter the following command:

> text(locator(1),”1977”,cex=1.2, adj=0, col=2)

After entering this, click (with the left mouse button) on the plot at the exact point you would like this left-justified text string “1977” to begin.

A common task in data analysis is to plot a smooth function f(x) versus some values x. Often this smooth curve is superimposed on a scatter plot of points. The curve()function in R is handy for this task. If we use least squares to fit a quadratic function to the scatter plot we have created, the equation of the fitted quadratic is 1.66+5.56\*x -0.46\*x^2. Add the fitted quadratic to the plot like so:

> curve(1.66+5.56\*x-0.46\*x^2,from=0.5,to=2.8,add=T)

> text(0.5,14,”With Fitted Quadratic”, adj=0, col=2)

Note: you must use the symbol x in the expression for the curve rather than the name of the variable plotted on the axis. The first argument to curve() can be an expression in x like the one above, or a function name. For example, here is a plot of the standard normal density curve:

> curve(dnorm,from=-3,to=3,lwd=3,col=4)

Presentation-quality and publication-quality graphics often require a high degree of user control over the plotting package. This can be managed in R by putting the plot together piece by piece. You will probably want to write a script or function (Chapter 12) to do this, since the script or function can be changed easily and then re-called to modify the graph when needed. Whenever I write a paper I create functions for every figure, called figure1(), figure2(), etc.

Often the first step in a plot which is put together piece-by-piece is the construction of an “empty plot”, one which exists only to set up an axis system to locate add-ons:

> plot(xvec,yvec,

type=”n”,xaxt=”n”,xlab=””,yaxt=”n”,ylab=””)

This statement seems to create an empty box (you can opt not to have the box, too) but sets up a coordinate system necessary to include the values contained in xvec and yvec. The default x-axis and its label are suppressed by the options xaxt=”n”, xlab=””; these can be added after-the-fact with (e.g.) an axis() statement. After an “empty plot” statement like the above, you can add points, curves, line segments, symbols, arrows, text strings, and so on to your heart’s content using the created axis system.

This “empty plot” stepwise technique is one way to make plots with multiple curves and/or plotted points of different types. There is also a function **matplot()** that can superimpose multiple curves or scatter plots by plotting all columns of a matrix, but it seems less flexible at times than what is needed.

An easy way to make many scatterplots quickly is the scatter plot matrix, whose generic call is:

> pairs(numeric.matrix)

Try this on four of the state.x77 variables to explore relationships between education, murder rate, and life expectancy:

> pairs(statex77.dfr[,3:6])

Crowded as it is, this plot allows us to quickly examine every pairwise relationship.

Another useful graphical technique related to the scatterplot matrix is the conditioning plot or “coplot”. In a coplot, a variable y would be plotted against another variable x, separately for each value of a “conditioning” variable. It has a somewhat unusual syntax…here is a call for the Murder vs. Illiteracy data, conditioning on state.region:

> coplot( Murder ~ Illiteracy | state.region)

A data frame containing the variables must usually be attached, or can be referenced as part of the function call using the data= argument. Sometimes the key for a coplot can be a bit misleading – the one just made has the South region in the lower right.

## 11.2 Multiple Plots on a Page

The easiest way to construct multiple plots on a single page is through the mfrow and/or mfcolparameters in par(). For example, suppose we desire two plots side-by-side on a page (i.e. in 1 row, with 2 columns):

> par(mfrow=c(1,2))

Then proceed with both of your plots. When R encounters the second plot statement, it assumes the first plot is finished; any additions after that point will be made to the second plot. If you would like 6 plots laid out in two rows of three plots each, use par(mfrow=c(2,3))or par(mfcol=c(2,3)). The difference between these two choices is the order in which plots are created - by row for mfrow**,** by column for mfcol. Below is a slightly complicated example that also demonstrates the use of **for** loops and yields a useful souvenir. Type these commands very carefully, exactly as shown. The result is shown on the next page. You may want to make a hard copy of this figure for future reference, if you have a printer handy. My copy is hanging on my office wall.

> par(mfrow=c(1,2),pty=”s”,mai=c(0.5,0.4,0.4,0.5))

> plot(1:25,1:25,type=”n”)

> for (i in 1:25) points(i,i,pch=i,cex=0.7)

> title(“the 25 pch symbols”)

> plot(1:6,1:6,type=”n”)

> for (i in 1:6) lines(c(1,6),c(i,i),lty=i,lwd=2)

> title(“the 6 lty line types”)

The graphical parameter mai in the **par** statement sets the size of the bottom, left, top, and right margins in inches for each plot, respectively, since the default margin sizes in R often seem too large (try the above code without the mai specification).

The commands including the word for are loops…the first one executes 25 statements. It is equivalent to entering the following commands in order:

> points(1,1,pch=1,cex=0.7)

> points(2,2,pch=2,cex=0.7)

> points(3,3,pch=3,cex=0.7)

**...**

and so on.

**Side-by-side plots of 25 plotting characters and 6 line types.**

## 11.3 Three-Dimensional plots

Three-D plots represent the values of a numeric matrix as a third dimension, with the row and column indices (or numeric vectors of the same lengths) representing the first two dimensions. There are at least four types of 3D plots in R: contour(), filled.contour(), persp() and image(). The most basic call is of the form (e.g. for contour)

> contour(zmatrix)

Let’s try this for the built-in numeric matrix of approximate topographic information on the Maunga Whau volcano in Auckland, New Zealand:

> data(volcano)

> contour(volcano)

Try the filled.contour(), persp() and image() plots on this data, too. To modify colors used, see the help file for the heat.colors(), topo.colors(), terrain.colors(),and/or palette() functions.

Unless more meaningful values are provided, these functions use the matrix’ row and column numbers to determine the x- and y-values, scaling these numbers to lie between 0 and 1. If meaningful x- and y-values are available, e.g. latitude and longitude (or “northing” and “easting”) these can be provided in the call, something like this:

> contour(xvec, yvec, zmatrix)

Note: it shouldn’t be a problem with R, but when constructing any contour or 3D plot one should always make sure that the values in the top left corner of the contoured matrix correspond to what you want to have at the bottom left corner of the contour plot. For some contouring packages this is not true, in which case you should contour the transpose of the matrix or reverse the ordering of the rows of the matrix prior to contouring.

All the 3D plot functions accept graphical parameters to “pretty them up”; all can be enhanced using the created x-y coordinate system and adding text, symbols, etc. The **contour** function can also output a list of contour-curve coordinates which might be useful. Contours can also be added to an existing plot.

**11.4 Modern graphics packages**

The latticelibrary provides trellis high-resolution graphics, a powerful and elegant high-level data visualization system with an emphasis on multivariate data. It includes a more flexible set of options for multiple plots and three-dimensional plots than what is discussed here. Trellis is very powerful for co-plots and figures with multiple plots, too. The develop of the lattice system in R, Deepayan Sarkar, wrote a book entitled Lattice: Multivariate Data Visualization with R (2008, Springer) that remains a good recent reference for trellis graphics in R (. To load the trellis graphics package, enter

> library(lattice)

The ggplot2 library is part of the “tidyverse”, a unified set of tools developed by Hadley Wickham to streamline R code; the popularity of these tools has exploded in recent years. It is not unusual to be asked for help with R code that looks nothing like the R code I first learned—that’s because I have wandered into the tidyverse. Developing graphs in ggplot2 is so very different from the usual R graphics that you may find yourself choosing one platform or the other; likewise with the tools in dplyr, another tidyverse package, for data management. The seminal text is *R for Data Science: Import, Tidy, Transform, Visualize and Model Data* (2017, O’Reilly) by Hadley Wickham and Garrett Grolemund.

## 11.5 Interactive Graphics and Exploratory Data Analysis

Reconstruct the basic plot of Murder vs. Illiteracy:

> plot(Illiteracy,Murder,xlim=c(0.3,3),ylim=c(0,15))

R offers some capability to interact with plots; for example, if we are curious about an unusual point on a plot, we may be able to identify it. Suppose the plot’s points are given by vectors xvec and yvec, and there is another vector of labels for these points (of exactly the same length – and all three vectors must be sorted in the same way!). A generic call to the identify() function, after the scatter plot has been constructed, looks like:

> weirdos = identify(xvec,yvec,labels)

After entering this command, using the left mouse button, click on a point on the plot and its label should appear near the point. You may identify as many points as you like in this way. When finished, click the right mouse button and choose Stop. The created object, here called weirdos, is a vector containing the indices of the identified points, in the order identified. These indices could be used to extract these points from xvec and yvec or a data frame or matrix having the same number of rows as labels.

For example, suppose we are curious which state has the highest murder rate. Let’s find out.

> mypoints = identify(Illiteracy,Murder,state.abb)

Now click under the point with the highest 1977 murder rate. Continue clicking as long as you want. When finished, click the right mouse button and choose Stop. Now investigate the created object mypoints :

> mypoints

> state.abb[mypoints]

The original S package had even more built-in interactive capabilities than those discussed above, and Splus has retained some of these, but R unfortunately has not. These capabilities include “brushing” and “spinning”. In brushing, a scatter plot matrix is first constructed, and then points on any given plot are identified using the mouse. As they are identified on any one plot, the same points are highlighted on every other plot. In spinning, three variables are plotted on x-y-z axes, and using the mouse the user is capable of spinning the axes to give the illusion of a three-dimensional point cloud. What actually happens is that 2-dimensional projections of the three-tuples are computed quickly and plotted, then recomputed and replotted, and the viewer “fills in” the illusion of three-dimensional form in a fashion similar to what happens with 3-D glasses. These capabilities have been removed from the base package for R, but can be added as optional packages, including ggobi, loon, and plotly, which supports ggplot2.

# 12. AN INTRODUCTION TO SCRIPTS AND FUNCTIONS

R will be most useful to you when you become comfortable writing your own programs – programs in R are called scripts or functions.

## 12.1 Scripts

A script is just a series of R commands, pretty much the same as would be entered at the prompt, except without prompts and typed into a text file. When you run the script, or a selected part of the script, the commands are executed just as they would be by entering them at the prompt one at a time. RStudio has made it much easier to manage scripts interactively during an R session.

Here is a script to make the Murder rate vs Illiteracy figure from Chapter 11: Go to + ⇒ R Script. In the new tab in the upper left-hand frame, enter the following commands (no prompts)…if you cut and paste from Word, you may have to re-type the quotes:

data(state)

state.dfr = data.frame(state.abb,state.region,state.x77)

rm(state.abb,state.region,state.x77)

names(state.dfr)

attach(state.dfr)

plot(Illiteracy,Murder,las=1,lwd=2,cex=1.2,pch=19,xlim=c(0.3,3),ylim=c(0,15),xlab="percent Illiterate in State", ylab="Murder rate per 100,000")

title("Murder vs. Illiteracy")

text(2,3,"The 50 United States",cex=1.2,adj=0,col=2)

detach(state.dfr)

rm(state.area,state.center,state.dfr,

state.division,state.name)

Before running the script, click the Save icon and give your script a name - it will be saved as an R file with the .R extension. Run the commands by selecting them and typing Ctrl-R (on a Mac, type Command-Return). You can also run only a part of the script – just select the subset of desired commands and use Ctrl-R.

Since a script will run pretty much exactly as the same series of commands would run if issued one at a time, a disadvantage of using scripts is that any assignment will create a new object in the workspace. It is important then that the last few commands of the script remove all script-created clutter and detach data frames that you are finished working with. What would happen if you ran the script above repeatedly without the detach(state.dfr) command?

## 12.2 Writing functions

A function in R can do similar work as a script but does not save objects created along the way. Ideally, a function is a program that performs a generic task with minimal user intervention - a tool that can be used repeatedly in many settings.

As an example, the next page shows a text file which is a home-written function to perform a two-sample t-test and confidence interval for a difference of population means (for those of you who have had elementary statistics, this is the t-procedure that assumes independent samples and uses the pooled sample variance). We will get this function into R in three steps:

1. Write or paste the function commands carefully in a text file using, say, Wordpad. Note that the function begins with the word function followed by parentheses containing argument names and, if desired, their default value assignments. In this example, the default value for alpha is 0.05, and for main (the plot title) is a blank. Immediately following the closing parenthesis after the argument list is a left brace. The body of the function (its statements) must lie between this brace and its closing right brace. The first few lines of the function are usually comments, statements starting with # signs, explaining briefly what the function does, what it accepts as its input arguments, and what kind of object, if any, it returns. When you feel fairly certain that the function is correctly written, select the text and copy it to the clipboard.
2. Use the fix() function to create an “empty” function with the appropriate name (suggestion: use verbs in function names to help find them in the workspace…or an extension like “fct”):

> fix(twosamplet.fct)

This opens a text editor window. At this point the window should show only the

words “function() {}”; delete these (without copying them to the clipboard)

and paste your own function text in their place. Then select Save.

1. (a) If the R prompt returns with no error messages, the function has (as far as R can tell at this point) no obvious syntax errors and now resides in the workspace as a working function ready to test. Check it by issuing the ls() command and typing its name at the prompt :

> twosamplet.fct

(b) If, however, when the fix() session ends, you receive an error message similar to this one:

Error in edit(name, file, editor) : An error occurred…

(etc etc ...some sketchy details will be provided)

use a command like

x <- edit()

to recover

This means that the R syntax checker has found fault with your function’s syntax - for example, there may be left parentheses or braces without matching right parentheses or braces. The original function (in this case twosamplet.fct) has therefore *not* been saved in the workspace yet. Be very careful at this step or you will lose the work done during the most recent fix() session. To return to the text file you were just working on, issue the command (if the function name is twosamplet.fct)

> twosamplet.fct = edit()

Make appropriate changes and save the results as a text file, using File ⇒ Save As. Then go to File, Save, and Exit. If the R prompt returns with no error messages, the syntax errors have been repaired. If you get the “Error in edit” message again, all syntax errors in your function have not been fixed.

As an exercise, create a text file twosamplet.txt from the code shown below, and try to get it into R as a function. Also, try to understand what each command of the function is accomplishing, step-by-step.

The twosamplet.fct function:

#################################################

function(yvec,trtvec,alpha=0.05,main="") {

#################################################

# A function to compute a two-sample t-test and confidence # interval (equal-variance, independent samples). yvec is # a numeric vector containing both samples' data. trtvec

# is a vector or factor, same length as yvec, of treatment

# identifiers for the data in yvec. A boxplot comparing

# the treatments' data is constructed. Output is a one-row

# data frame reporting the results of the test and the

# confidence interval.

##################################################

trtvec=as.factor(trtvec)

boxplot(split(yvec,trtvec))

title(header)

ybar=tapply(yvec,trtvec,mean)

varvec=tapply(yvec,trtvec,var)

nvec=table(trtvec)

error.df=nvec[1]+nvec[2]-2

pooled.var=((nvec[1]-1)\*varvec[1]+(nvec[2]-1)\*varvec[2])/error.df

diff12estimate=ybar[1]-ybar[2]

stderr=sqrt(pooled.var\*((1/nvec[1])+(1/nvec[2])))

tratio=diff12estimate/stderr

twosidedP=2\*(1-pt(abs(tratio),error.df))

tcrit=qt(1-alpha/2,error.df)

lower=diff12estimate-tcrit\*stderr

upper=diff12estimate+tcrit\*stderr

out=data.frame(diff12estimate,stderr,tratio,twosidedP,lower,upper,alpha)

out

}After the function exists in the workspace, for practice, open the function as though you wish to modify it further:

> fix(twosamplet.fct)

Now, remove a parenthesis or a brace somewhere, and Save. This should cause an error as in case 3b above. Try to get back to the original function using the edit()command as discussed above in 3b, replace the missing brace / parenthesis, and save the function again.

Some general rules and comments about functions:

* If the function will return anything to the calling program, it can return one and only one object, though this object may be a list. The one returned object is named in the last statement of the function, before the closing “}” brace.
* The function statements can operate on data passed as arguments (using the internal function argument names, not the names of the objects as they exist outside the function), or on objects it creates, or on objects that exist in the workspace or search path when the function is called. Any objects created by the function will not be saved when the function terminates unless they are passed back in the final statement. However, changes to the search path seem to be preserved even after the function terminates (e.g. if the function creates and attaches a data frame, that data frame seems to still be present when the function terminates). *Therefore, if your function attaches any data frames, you should add statement(s) to detach them near the end of the function body.*

To test out your new function for doing two-sample t-tests, what better data set to use than the one used by W.S. Gossett (“Student”) in his 1908 article on the t-distribution? Load the data:

> data(sleep)

By entering help(sleep), find that this is a 20-row data frame containing two columns. The group column identifies a “soporific treatment” applied to each subject; it is a factor object with levels “1” and “2”. The first column extra gives the extra sleep in hours over some baseline amount for that subject. Before calling twosamplet.myfct, check the workspace contents. Then call it as follows:

> results = twosamplet.fct(sleep$extra, sleep$group)

Notice the function has created boxplots of the two groups’ data. Its numerical output can be viewed by entering the name of the created object:

> results

If no name had been assigned to the function output, these results would have been printed on the screen without being saved in the workspace.

## 12.3 Debugging your function with the **browser()**

When a function is successfully saved into R this does not necessarily mean that it is error-free. There may be syntax errors which are not detected until run time, or there may be those wonderful logical errors under which the function will run just fine but produce erroneous output. R has a very nice tool for debugging functions known as the **browser**. To use it, edit your dysfunctional function with, e.g.

> fix(my.fct)

and insert the command

browser()

at a strategic spot – usually the spot just before the trouble seems to occur. Save and quit the function; then try to run it again. This time, the browser will stop the function execution at exactly the spot you inserted it and give a browser prompt like this:

Browse[1]>

From this prompt, commands can be issued to examine any objects that the function has created up to this point. One can also look at the function code itself just by entering the function name, and copy and paste statements from the function code following the browser command and see where the crash actually occurs. Usually the problem(s) will be revealed by inspection of objects created just before the trouble spot in the function, perhaps checking their attributes or dimensions or making a quick plot or two. When finished browsing around, enter “c” at the browser prompt to continue executing the function:

Browse[1]> c

Following this the function will resume execution at the next command after browser(). Or, enter Q,

Browse[1]> Q

and R will quit executing the function altogether (note the capital Q). Or, enter n,

Browse[1]> n

and this will bring up the next command after the browser() command inside the function for inspection. Using this feature, the function code after the browser()statement can be executed one line at a time.

If there are loops in the function, and if problems seem to occur inside a loop, it may be inconvenient to stop the function’s execution on every iteration of the loop. In this case, use an if()clause to stop the function execution only under some condition. For example, suppose the loop does a calculation to produce a numeric scalar called yvalue, and you suspect that the bug inappropriately produces missing values. Insert the following line immediately after the yvalue calculation in the function:

if (is.na(yvalue)) browser()

Entering Qat theBrowse[1]>prompt is particularly valuable to exit from a loop that would iterate 1000 more times, each time calling the browser !

When finished browsing the function, enter Q to exit the browser. Re-edit the function with, e.g.

> fix(my.fct)

and make final changes, which would usually include removing the browser() command, or making it a comment with the # sign, or moving it further down the function code if more errors are expected.

Once the function seems to be running well, make a text file copy. You are required to give me $1 every time you do this and forget to remove the browser() command.

## 12.4 More function examples

1. Let’s play cards…

poker = function() {

# a function to deal a poker hand

# first, create a card deck

suit = rep(c("S","H","D","C"),13)

suit = sort(suit)

denom = c("A",as.character(2:10),"J","Q","K")

denom = rep(denom,4)

carddeck = cbind(denom,suit) # a 52x2 character matrix

#

# shuffle and deal five (randomly select 5 from the deck)

cardnums = sample(52,5)

mycards = carddeck[cardnums,]

mycards

}

2. A “statlet” : a function to teach least squares. Thanks to Charles Champion of the Stat 540 class fall 2015 for suggesting readline

least.squares = function (n,corr=0.8) {

# a function to illustrate a least squares line fit versus eyeball fitting

# The MASS package must be loaded to use this function

#

# first, generate and plot n points with the correlation

Sigma = matrix(c(1,corr,corr,1),2,2)

mypoints = mvrnorm(n,mu=c(0,0),Sigma)

x = mypoints[,1]

y = mypoints[,2]

readline("push enter, then choose two locations to define your eyeball line of best fit.")

plot(x,y,pch=1,cex=1.5,xlab="",ylab="")

title("Try Ur Luck in Beating the Least Squares Fit")

# pick points for an eyeball line

eb = locator(2,type="l",col=2,lty=2,lwd=2)

#eb is a list giving the eyeball line endpoints

slope = (eb$y[2]-eb$y[1])/(eb$x[2]-eb$x[1])

yhat.eb = eb$y[1] + slope\*(x-eb$x[1]) # predicted values using eyeball fit

SSE.eyeball = sum((y-yhat.eb)^2)

#text(0,0,paste("eyeball fit SSE=",signif(SSE.eyeball,4)),adj=0,col=2)

# fit the least squares line and get the minimum SSE

lmout = lm(y~x)

abline(lmout,col=3,lty=1,lwd=2)

SSE = sum(residuals(lmout)^2)

legend1 = paste("eyeball fit SSE=",signif(SSE.eyeball,4))

legend2 = paste("least squares SSE=",signif(SSE,4))

legend("topleft",c(legend1,legend2),lty=c(2,1),col=2:3,lwd=c(2,2))

title(sub="You Lose! wanna try again?")

}