13.1 Alternative *R* Tutorial 7

Introduction to Computational Science:
Modeling and Simulation for the Sciences, 2nd Edition
Angela B. Shiflet and George W. Shiflet
Wofford College

Introduction

We recommend that you work through this tutorial with a copy of *R*, answering all Quick Review Questions. *R* is available from http://www.r-project.org.

This Alternative Tutorial 7 has no prerequisite and is for those starting their use of *R* with Chapter 13. For those who have worked through *R* Tutorials 1-6, refer instead to Tutorial 7, *RCTTutorial7*. *RCTTutorial7* and *R RCTTutorial7Alt* prepare you to use *R* to complete projects for this chapter. The tutorial also gives examples along with Quick Review Questions for you to do with *R*. Execute all commands to view the results of the examples.

From Tutorial 1

Getting Started

Make sure that the R application is open, and that you have access to the R Console window. For the following material, at a **prompt** of >, type each example; and evaluate the statement in the Console window. To evaluate a command, press ENTER. In this document (but not in R), input is in red, and the resulting output is in blue.

We start by evaluating 12-factorial (also written "12!"), which is the product of the positive integers from 1 through 12. We can use R's built-in *factorial*() function[1] to accomplish this by typing "factorial(12)." After pressing ENTER or RETURN for the first computation, R returns the result, as the following prompt (>), input (in red), and answer (in blue) indicate:

```
> factorial(12)
[1] 479001600
```

Notice that the number "1" appears in brackets before the answer. You can ignore this fact for now; for the curious, this is because in R, all values are *vector* instead of *scalar* quantities, and so what *factorial*(12) actually produces is a vector with one element (namely, 479001600). For 25!, the following output appears in exponential notation with e+25 indicating multiplication by 10^{25}, so that the value is 1.5511×10^{25}:

```
> factorial(25)
[1] 1.551121e+25
```

1 An R "function" can be thought of similarly to a mathematical function: it gives an output for certain input(s). When referring to R functions, we typically include an empty pair of parentheses immediately after the function name, as in sin() or cos(), to make clear that it is a function rather than a simple variable.

R Programs

Many simple operations can be carried out directly at the R console, but often it makes more sense to write an R program (sometimes also called a "script") that contains multiple different commands organized to carry out some more complex operations. You can open a new script in Windows or Mac by using the "***New***" command from the "***File***" menu. (In Linux, you will simply use your favorite text editor, making sure to save the file in the R working directory.)

The first part of an R program is often an explanation of what the program does, for instance:

```
# hurricaneSimulation.R
# Simulate a category-5 tropical storm to 100-km accuracy
# Filbert Lewis
# Jan.10, 2015
```

The lines above are referred to as **comments**. Frequently, and especially when writing R programs (see below), we need a "comment," or explanation of material, which we do not want R to execute. We begin a comment line with a **# sign** (pronounced "pound" or "hash" or "tic-tac"), as above. R will ignore everything on that line, considering it fit for only humans to read.

To save a file, we use the familiar "***File***" menu, entering a file name to which R appends the extension **.R**. ***Save often***, particularly before you print, do a long calculation, or display a graph.

To execute an entire R program, use the ***source*** command from the R console, like this:

```
> source("filename.R");
```

For example, if you saved a program under the name "hurricaneSimulation," you would run it like this:

```
> source("hurricaneSimulation.R");
```

Sometimes, it's helpful to have R print out the evaluation of every command in the .R program file as it runs. To do this, we can include the text ",print.eval=TRUE" after the name of the file, like this:

```
> source("hurricaneSimulation.R",print.eval=TRUE);
```

Note that R program files normally don't have spaces in them, since this can cause problems. It's a good idea to capitalize the first letter of each successive word, as in the hurricaneSimulation example, above. (This is called "camel case" for obvious reasons.)

If, when you type the *source*() command, you get an error message saying that R could not find the file (sometimes this error message says something like "cannot open the connection"), one of three things is probably wrong: (a) you mistyped the filename, or (b) you forgot to type the ".R" extension as part of the source command, or (c) the folder/directory that R is using is different from the folder/directory that contains the file. To diagnose the latter, type the command "*getwd*()" in the R console (or under the ***Misc*** menu, select ***Get Working Directory***), and press ENTER. This command will print out the name of the folder/directory that R is looking in. If it does not match the name of the folder/directory you saved your file in, then this is the problem. You have two choices: either tell R to change to the correct folder/directory, or else move your file to the folder/directory R is using. The latter is probably simpler, although if you do this you

will need to remember to save all of your files into that new folder/directory instead of your original one. To do the first option, under the *Misc* menu, select ***Change Working Directory…***, maneuver to the desired directory, and select *Open*.

Numbers and Arithmetic Operations

Scalar multiplication in R is indicated by an **asterisk** (*) (also called a "star" or "splat.") Thus, to multiply 5 times 18, we write `5*18`, not 5(18) or 5x18. The addition, subtraction, and division operators are `+`, `-`, and `/`, respectively. The operator `%%` (called "modulus") will give the remainder after dividing two integers. (For instance, `11%%3` gives `2` because 11 divided by 3 is three, with a remainder of two.) An expression can be raised to a power using a **caret** (^) (also called a "hat") as follows:

```
> 3 * (5/8 - 1.25)^2
```

R has numerous built-in functions, such as `sin()`, and built-in constants, such as `pi` representing π. A function usually has one or more inputs, also called "arguments" or "parameters." To use a function, you type the name of the function, followed by parentheses around the argument(s). (Multiple arguments should be separated by commas.) For instance, to compute 5 sin(π/3):

```
> 5 * sin(pi/3)
```

(Again, the asterisk for multiplication is mandatory.)

Variables and Assignments

We can employ **variables** to store values for future use. Variable names must begin with a letter, and be composed of letters, digits, periods, and underscores. Variable names are case-sensitive in R, meaning that variables named "length," "Length," and "LENGTH" will all refer to *different* variables. As mentioned above, the use of camelCase is recommended for variable names comprised of multiple words.

We can **assign** a value of an expression to a variable this way:

variableName* = *expression

This sets the value of the variable called *variableName* to whatever *expression* evaluates to. For example, the following assignment sets the variable called *lengthOfBridge* to 15:

```
> lengthOfBridge = 15
```

R calculates the value of the expression on the right, such as 15, and then assigns the value to the variable on the left, such as *lengthOfBridge*. If we subsequently refer to *lengthOfBridge*, R replaces *lengthOfBridge* with its value (15, or whatever it might have been set to subsequently.)

Note that the statement "`lengthOfBridge = 15`" is *not* an **assertion** that the value of the variable *is* 15, but rather a **command** that instructs R to *make* the value be 15. For this reason, the use of the equals sign in this syntax is somewhat misleading, and some R programmers prefer to substitute the key sequence "**<-**" instead, as in:

```
> lengthOfBridge <- 15
```

(These two statements are equivalent.)

If we use a variable before it has been assigned a value, R returns an error message, such as "Error: object 'silly' not found".

We can **clear** a variable (*i.e.*, erase its value) by using the "*rm*" (for "remove") command. The argument is the name of the variable we want to clear. For instance, if we have a variable x whose value is 17.5, and we no longer want there to *be* a variable x at all, we can simply type:

```
> rm(x)
```

At any time, we can see what variables are defined via the "*ls*" (for "list") command:

```
> ls()
```

A common technique in R programming is to use a variable to cumulatively sum a quantity. In these situations, we need the old value of the variable to compute the new value of the same variable, as in:

```
> numberOfBirths = numberOfBirths + birthsThisYear
```

A variation of this technique is to "count" some continuously incrementing quantity over time. For instance:

```
> day = day + 1
```

Again, we stress that these statements do not *declare* that a variable is equal to itself plus (say) one, but rather *instruct* R to set the variable to a new, higher, value.

In R, a variable can hold a whole sequence of numbers, rather than just a single number. A sequence of numbers is called a **vector**. One way to create a vector is by **c**ombining several individual numbers into a vector variable using the "*c*()" function. As an example, if we measured the heights (in meters) of various trees in a forest, we could create a variable called heights that could hold all of the values this way:

```
> heights = c(21.6, 22.5, 19.8, 20.5)
```

The heights variable now holds four different numbers, one for the height of each tree we measured. Typing the name of the variable shows all of its elements:

```
> heights
[1] 21.6 22.5 19.8 20.5
```

To extract the value of a *single* tree's height, we use the **square bracket notation** to specify the **index** of the value we want:

```
> heights[1]
[1] 21.6
> heights[3]
[1] 19.8
```

There are other ways of quickly creating sequences of numbers. The function "*seq*()" creates a vector with numbers in a **seq**uence, given a starting and ending number:

```
> years = seq(1994,2011)
> years
[1] 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008
[16] 2009 2010 2011
```

(Note that there are so many values in this vector that the elements "wrap around" to a second line. The number in brackets at the start of each line ([1], [16]) give the index of the element at the start of that line, for easy reference.)

Also, the **colon notation** can be used as a more compact way of defining these vectors. The expression "1994:2003" is a shorthand for "all the integers (whole numbers) between 1994:2003."

```
> years = 1994:2003
> years
[1] 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003
```

Returning to the *seq* function, we can also add a third argument to specify the **increment** (or "**step**") between values, if we want the values to increase by something other than 1. For instance:

```
> years = seq(1992,2012,4)
> years
[1] 1992 1996 2000 2004 2008 2012
```

Variables in R are not always numeric; they can also contain text data, known as "character strings." For instance, one could declare a variable this way:

```
> myName = "Stephen"
```

Note the use of double-quotes to specify character strings. If we had omitted the quotation marks in the previous example, R would have given an error message because it would have been looking for a *variable* named "Stephen" to assign to the variable myName, instead of treating "Stephen" as a value.

Quick Review Question 9 Create a new script / program file. Save the file under the name *RCTTutorial7Alt.R*. In the file, preface this and all subsequent Quick Review Questions with a comment that has "QRQ" and the question number, such as follows:

```
# QRQ 1
```

Paste your code from the R Console into this file. Use the *c*() function to create vector variables for each of the following:

a. A vector called *ages* with values 19, 21, 21, and 20.

b. A vector called names with values "Ruth," "Callixte," and "Talishia." (Note: this should be a vector of *three* values, the first of which is "Ruth" and the last of which is "Talishia." It should *not* be a vector of just one value, whose value is "Ruth, Callixte, Talishia.")

User-defined functions

Frequently, we wish to define our own functions that we can use again and again. As with variable names, by convention, we begin the name of such a **user-defined function** with a lowercase letter.
Suppose we wish to define the function $sqr(x) = x^2$ in R. At the console, we can type:

```
> sqr = function(x) x^2
```

This tells R that a new function variable called *sqr* is to be set equal to a function of a variable x , and that the value of $sqr(x)$ should be x^2.

We can "call" (or "invoke") the function on a particular value just like we did for the built-in functions *factorial*(), *c*(), and *seq*():

```
> sqr(14)
[1] 196
```

User-defined functions longer than this are typically written in an R program file as part of a larger program, rather than typed into the console. However, the syntax is the same.

Quick Review Question 11

a. Define a function $quick(x) = 3\sin(x - 1) + 2$. (Be careful to remember how to express multiplication in R. Typing "3sin(x-1)+2" will not work.)
b. Evaluate the function at x = 5.

Online documentation

R has an extensive help system built in to the console that is very easy to use. To find out information about a particular function (whose name you know), you can type "**?**" followed by the name of the function. For instance,

```
> ?sin
```

brings up a page of information about sine, as well as other related trigonometric functions. (Linux users should press "q" (for "quit") to return to the R console.) Alternatively, from the ***Help*** menu, we can select ***R Help***, and then an option, such as ***Search Engine & Keywords***.

If you need help on a topic but aren't sure what the exact name of the R function is, you can also *search* the help system for a word or phrase. To do this, type "**??**" (*two* question marks) followed by your search string. You will be presented with a list of all the R help topics that contain that string. For instance, suppose you want to perform a statistical test of proportions (this is a statistical test to compare whether the proportions in two groups are significantly different from each other; for instance, whether the proportion of 15-year-olds who regularly play videogames differs from the proportion of 18-year-olds who do.) You aren't sure of the name of the R function, so you guess "proportion." Typing "?proportion," however, gives a "no documentation" message. So you decide to search the help system by typing:

```
> ??proportion
```

Scanning the results, you see that stats::prop.test is a function that will perform a "Test of Equal or Given Proportions." This tells you that prop.test() is the function you want to use. (Note that the "stats::" prefix is a **package name**. Packages in R are related sets of functions and data files that are grouped together. You don't need to type the name of the package to use a function, but you can.)

To get specific help about how to use the *prop.test()* function, you could then type:

```
> ?prop.test
```

(with just one question mark.)

Quick Review Question 12 At the R console, access the help page for the built-in function *log10*. Scroll through this help page, then return to the R console. Paste the code to do so *as a comment line* in *RCTTutorial7Alt.R*.

Quick Review Question 13 Suppose you do not know the R command to perform a standard deviation. Typing "?standard" and "?deviation" do not yield any results. Use the help system to *search* for the word "deviation" and see if you can discover the name of the function. Record your findings in *RCTTutorial7Alt.R*.

Displaying

Sometimes, particularly when doing error checking, we wish to display intermediate results. To do so, we can employ the *cat* function. ("cat" stands for "con**cat**enate.") *cat*'s arguments are a list of character strings that should be concatenated (or "stuck") together. In order to get cat to print a **newline** (carriage return / line feed) after it prints a result, the two-character sequence "**\n**" must be added. For instance, the following command prints a friendly hello:

```
> cat("Why hello there",myName,"!!\n")
Why hello there Stephen !!
```

Note that this command provided *three* character strings to stick together in the output: (1) "Why hello there," (2) the value of myName (which is "Stephen" or whatever that variable's current value is), and (3) a string with two exclamation marks and the newline sequence.

Numeric values, of course, can also be printed:

```
> gpa = 3.5
> year = 2011
> cat("Your GPA in",year,"was",gpa,".\n")
```

Your GPA in 2011 was 3.5.

Looping

It is often advantageous to be able to execute a segment of code a number of times. For example, to obtain the velocity for each integer time ranging from 1 to 1000 seconds, it would be inconvenient for the user to have to execute one thousand statements assigning a time and computing the corresponding velocity. Some method of automating the

procedure is far more preferable. A segment of code that is executed repeatedly is called a **loop**.

Two types of loops exist in R. When we know exactly how many times to execute the loop, the **for loop** is a good choice. One form of the command is as follows:

```
for (i in min:max) {
    expr
}
```

Recall that the notation *min*:*max* is a shorthand for *seq*(*min*,*max*), meaning that it represents a vector containing the *entire* sequence of numbers from *min* to *max*, including both end points. In this loop, then, the **index** or **loop variable** is *i*, and *i* takes on integer values from this sequence. For *each* value of *i*, the computer executes the body of the loop, which is *expr*, the statements between the curly braces.

For example, suppose, as the basis for a more involved program, we wish to increment a variable called *dist* (for "distance") by 2.25 seven times. We initialize the variable to 0. Within a for loop that executes 7 times, we calculate the sum of *dist* and 2.25, and assign the result of the expression to *dist*, giving the variable an updated value. Because the for loop does not return a value and we are not printing intermediate values of dist, we display the final value of dist after the loop with the *cat*() function:

```
dist = 0
for (i in 1:7) {
    dist = dist + 2.25
}
cat(dist,"\n")
15.75
```

Quick Review Question 15 Write a segment of code to assign 1 to a variable *d*. In a loop that executes 10 times, change the value of *d* to be double what it was before the previous iteration. After the loop, type *cat*(*d*,"\n") to display *d*'s final value. Before executing the loop, determine the final value on your own so you can check your work. Then test your code, and paste it into the appropriate place in *RCTTutorial1.R*.

In the next loop, we increment *dist* by 2.25, compute time as $\frac{24.5-\sqrt{600.25-19.6dist}}{9.8}$, and then print out the distance and time for each time step.

```
dist = 0
for (i in 1:7) {
    dist = dist + 2.25
    t = (24.5 - sqrt(600.25 - 19.6 * dist))/9.8
    cat("For distance",dist,", time = ",t,"\n")
}
For distance 2.25 , time = 0.0935885
For distance 4.5 , time = 0.1909672
For distance 6.75 , time = 0.2926376
For distance 9 , time = 0.3992227
For distance 11.25 , time = 0.5115127
For distance 13.5 , time = 0.6305354
For distance 15.75 , time = 0.7576699
```

(A couple of notes on printing output: (1) notice that an extra space appears between the distance and the comma (",") in the above output. This is because by default, the "*cat*()" function pads its concatenated outputs with spaces. If you want to suppress these extraneous spaces, you can add the argument "sep=""" to the *cat*() command. (2) The time values are printed with many decimal places. To round to (say) two decimal places, you can use the *round*() function, passing the value you want rounded as the first argument, and the number of decimal places as the second argument. This modified program, and its output, appears below:)

```
dist = 0
for (i in 1:7) {
    dist = dist + 2.25
    t = (24.5 - sqrt(600.25 - 19.6 * dist))/9.8
    cat("For distance ",dist,", time = ",round(t,2),"\n",sep="")
}
For distance 2.25, time = 0.09
For distance 4.5, time = 0.19
For distance 6.75, time = 0.29
For distance 9, time = 0.4
For distance 11.25, time = 0.51
For distance 13.5, time = 0.63
For distance 15.75, time = 0.76
```

The previous example did not use the loop index in the loop's body. However, the following example of *cat*() and *for* with an index *i* displays both *i* and *i*-factorial (*i*!) with spaces between the values for *i*, going from 1 through 9:

```
for (i in 1:9) {
    cat(i," ",factorial(i),"\n")
}
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
```

To start the display with the value of 0!, which is 1, we indicate a beginning value of 0, as follows:

```
for (i in 0:9) {
cat(i," ",factorial(i),"\n")
}
```

Quick Review Question 16 For this question, complete another version of the code segment above that displays distance and time. In this version, do not initialize dist. Employ a loop with an index i that takes on integer values from 1 through 7. Within the loop, the value of dist is computed as 2.25*i*.

The code below is provided as a starting point in *RCTTutorial1.R*. After replacing each XXXXXXXX with the proper code, execute the program, and compare the results with the similar segment above.

```
for xxxxxxxxxx
```

```
    dist = xxxxxxxxxx
    t = (24.5 - sqrt(600.25 - 19.6 * dist))/9.8
    cat("For distance ",dist,", time = ",round(t,2)," seconds.\n")
xxxxxxxxxx
```

Quick Review Question 17

a. Define the function qrq17 to be ln(3*x* + 2). Recall that log is the R function for the natural logarithm.

b. Write a loop that prints the value of *k* and *qrq17*(*k*) for *k* taking on integer values from 1 through 8.

The other main type of loop that R supports is the **while loop**, which we will cover in future modules.

Plotting

We employ the "*plot*()" command for graphing two-dimensional functions. First, we establish a sequence of values for the independent variable, such as *t*. Recall that we employed sequences earlier as vector variables and in loop iterations. For example, "0:9" indicates the sequence 0, 1, ..., 9. For a smooth display of a graph, however, we need to plot additional points. To indicate a step size of 0.1, we could call the function like seq(0,9,0.1). The following statement assigns such a sequence to *t*:

```
> t = seq(-1,2,.1)
> t
 [1] -1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2 -0.1  0.0  0.1  0.2  0.3  0.4
[16]  0.5  0.6  0.7  0.8  0.9  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9
[31]  2.0
```

Now that we have an independent variable, we can use the *plot*() function. The basic form of the command gives the independent variable (such as *x* and the function (such as *g*()) to graph, as follows:

plot(*x*, *g*(*x*)**)**

For example, the following command graphs *cos*(*t*) with *t* varying from -1 to 2 by 0.1:

```
> t = seq(-1,2,.1)
> plot(t,cos(t))
```

R displays the resulting graphics in a figure window.

Quick Review Question 18 Graph $e^{\sin(x)}$ from -3 to 3. Experiment with different step increments, to see how it affects the graph display. Paste the code to generate this plot with 0.1 increments into *RCTTutorial7Alt.R*.

We can indicate additional options, and R revises the display to reflect our changes. For example, ***xlab*** and ***ylab*** options generate axes labels, which should appear in all scientific graphics. The following command will annotate the graph with sensible labels:

```
> plot(t,cos(t),xlab="t",ylab="cos(t)")
```

Quick Review Question 19 Adjust your answer to the previous Quick Review Question to label the *x* and *y* axes appropriately. Paste the modified code into *RCTTutorial7Alt.R*.

We can plot several functions on the same graph by using the *points*() function immediately after our initial *plot*(). The arguments are the same:

```
> plot(t,cos(t))
> points(t,.5*t+.4)
```

The second command in this sequence adds points to the plot according to the function $y(t) = .5t + .4$. We could change the shape (say, to the letter 'X') and color (say, to purple) of these new points by including those arguments to *points*() as well:

```
> plot(t,cos(t))
> points(t,.5*t+.4,pch='x',col='purple')
```

pch (plot character), ***col*** (color), *xlab*, and *ylab* (labels) are just a few of the many different parameters you can give to R graphics functions to format your output just the way you want it.

From Tutorial 2

Vectors

As you recall from R Computational Toolbox Tutorial 1, all variables in R are *vector* variables, meaning they have the capability to store more than one number at a time. One way we learned to create a vector was by using the *c*() function and listing its elements:

```
> dailyStockPrice = c(55.25, 56.5, 56, 57.75, 58.25)
> dailyStockPrice
[1] 55.25 56.50 56.00 57.75 58.25
```

Again, the "[1]" indicates that the element immediately to the right is element #1 of the vector. If we had made it longer:

```
> dailyStockPrice = c(55.25, 56.5, 56, 57.75, 58.25, 59, 59, 58.75, 55,
57, 51.25, 53, 48.75, 49.25, 50.25, 51.5)
> dailyStockPrice
[1] 55.25 56.50 56.00 57.75 58.25 59.00 59.00 58.75 55.00 57.00 51.25
53.00 [13] 48.75 49.25 50.25 51.50
```

the contents would have flowed onto a second line. The "[13]" indicates that the 48.75 is the 13th element.

We can also create vectors using the *seq*() function, by specifying a starting point, ending point, and increment amount:

```
> tideLevel = seq(15, 30, 1.5)
> tideLevel
[1] 15.0 16.5 18.0 19.5 21.0 22.5 24.0 25.5 27.0 28.5 30.0
```

Also remember that the colon operator (:) can be used as a shorthand for *seq*(), if the

increment amount is 1:

```
> lotNumbers = 56:70
> lotNumbers
[1] 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70
```

Vectors of any length can further be combined with *c*(). Consider:

```
> incomingTideLevel = seq(15, 30, 1.5)
> outgoingTideLevel = seq(30, 15, -1.5)
> tideLevel = c(incomingTideLevel, outgoingTideLevel, incomingTideLevel)
> tideLevel
[1] 15.0 16.5 18.0 19.5 21.0 22.5 24.0 25.5 27.0 28.5 30.0 30.0 28.5
27.0 25.5
[16] 24.0 22.5 21.0 19.5 18.0 16.5 15.0 15.0 16.5 18.0 19.5 21.0 22.5
24.0 25.5
[31] 27.0 28.5 30.0
```

Another useful function is *rep*(), which **rep**eats one value some number of times:

```
> adultTicketCosts = rep(15, 4)
> adultTicketCosts
[1] 15 15 15 15
```

Vectors created with *rep*() can of course be combined like any other:

```
> adultTicketCosts = rep(15, 4)
> kidsTicketCosts = rep(10, 7)
> seniorTicketCosts = rep(14, 1)
> ticketCosts = c(adultTicketCosts, kidsTicketCosts, seniorTicketCosts)
> ticketCosts
[1] 15 15 15 15 10 10 10 10 10 10 10 14
```

Interestingly, to extract elements out of a vector, you use another vector! The vector you provide contains *the index or indices of the element(s) you want to select*. For instance, to get element number 3 out of the ticketCosts vector, you type:

```
> ticketCosts[3]
[1] 15
```

To get elements 3 through 9, you do:

```
> ticketCosts[3:9]
[1] 15 15 10 10 10 10 10
```

And to get elements 2, 12, and 4 through 7, in that order, you do:

```
> ticketCosts[c(2,12,4:7)]
[1] 15 14 15 10 10 10
```

You can see what we did: first, create a vector containing 2, 12, 4, 5, 6, 7 by combining a 2, a 12, and the sequence from 4-7. Then, put that vector in the square brackets of *ticketCosts* to extract exactly those elements in that order. Part of R's great power comes from how easy it is to manipulate and combine vectors in this way.

Remember, too, that vectors lend themselves to *for* loops, since a variable can be used as the index into the vector:

```
for (i in 3:6) {
    cat("Ticket #",i," costs $", ticketCosts[i], ".\n", sep="") {
}
Ticket #3 costs $15.
Ticket #4 costs $15.
Ticket #5 costs $10.
Ticket #6 costs $10.
```

Quick Review Question 1 Use R functions to create vector variables for each of the following, copying your code into the *RCTTutorial2.R* file:

a. The numbers 47, 35, 22, and 10.

b. The numbers 1 through 12.

c. The numbers 4, 8, 12, 16, …, 84.

d. Eleven 3's followed by eleven 4's followed by twelve 5's.

e. 7, 6, 5, …, 1, 19, 2, 4, 6, 8, …, 30.

Matrices

A matrix is essentially a 2-dimensional vector. Each element in it has two indices rather than one, specifying an *x* and *y* coordinate.

You create matrices in R by first creating a vector with the data for the matrix, then calling the *matrix*() function and telling it with ***nrow*** how many rows the matrix will have. For instance:

```
> t = c(57, 61, 63, 64, 88, 89, 88, 86, 70, 81, 76, 76)
> temperatures = matrix(t,nrow=3)
> temperatures
     [,1] [,2] [,3] [,4]
[1,]   57   64   88   81
[2,]   61   88   86   76
[3,]   63   89   70   76
```

We've created a vector of raw data called *t*, then used it to create a matrix with 3 rows and 4 columns. Obviously, we have to tell R how many rows there are, otherwise it wouldn't know whether we wanted to create a 3 × 4 matrix (as we did), or a 4 × 3 matrix, 2 × 6 matrix, or even a 1 × 12 matrix. Equally obvious, we don't have to specify the number of columns, since if R knows how many elements there are, and how many rows,

the number of columns is fixed. (If we'd rather specify the number of columns than the number of rows, we can use the *ncol* parameter to *matrix*() instead.)

To extract an element from the matrix, we supply both row and column numbers:

```
> temperatures[2,3]
[1] 86
> temperatures[3,1]
[1] 63
```

If we want *all* the elements of a particular row, we just leave out the column number:

```
> temperatures[3,]
[1] 63 89 70 76
```

If this were a table of, say, the high temperatures in three different U.S. cities over the past four days, this would give us the temperatures for the third city. Note that the answer R gave us has a "[1]" to the left of the numbers, not a "[3]". This is because once the row is extracted, it is an ordinary stand-alone vector, and has no row number information any longer.

Similarly, we can leave out the row number:

```
> temperatures[,2]
[1] 64 88 89
```

and get the high temperature for all the cities on day 2. Again, note that the result is an ordinary 1-dimensional vector, which is therefore displayed horizontally (not vertically).

You can probably guess that R lets us combine vectors of indices:

```
> temperatures[2:3,c(4,1,3)]
     [,1] [,2] [,3]
[1,]   76   61   86
[2,]   76   63   70
```

This gives us the bottom two rows, and columns 4, 1, and 3 (in that order). This particular operation may or may not be useful, but it does show the flexibility of R's matrices. Note once more that the rows and columns of the resulting answer are labeled starting from 1, since the answer is a matrix in its own right.

Incidentally, just as the *length*() function told us the length of a vector, so the *nrow*() and *ncol*() functions give us the dimensions of a matrix:

```
> nrow(temperatures)
3
> ncol(temperatures)
4
```

Many times we will use matrices to represent the contents of a virtual map, and use for loops to iterate through them. To progressively move through a matrix requires a ***nested for* loop**, which means a *for* loop inside another *for* loop. This is required because for *each* row, you need to move through each of the columns. Here's what it looks like in

code:

```
for (r in 1:nrow(temperatures)) {
    for (c in 1:ncol(temperatures)) {
        cat("The high temp in city #", r, " on day #", c, " was ",
            temperatures[r,c], ".\n", sep="")
    }
}
The high temp in city #1 on day #1 was 57.
The high temp in city #1 on day #2 was 64.
The high temp in city #1 on day #3 was 88.
The high temp in city #1 on day #4 was 81.
The high temp in city #2 on day #1 was 61.
The high temp in city #2 on day #2 was 88.
The high temp in city #2 on day #3 was 86.
The high temp in city #2 on day #4 was 76.
The high temp in city #3 on day #1 was 63.
The high temp in city #3 on day #2 was 89.
The high temp in city #3 on day #3 was 70.
The high temp in city #3 on day #4 was 76.
```

Quick Review Question 2 Given the temperatures matrix, as defined above, write R expressions to return each of the following:

a. The element at row 3, column 3.

b. The element at row 2, column 1.

c. The entire 3rd row.

d. The entire 2nd column.

e. Columns 2 through 4 of rows 1 and 3.

Multidimensional Arrays

It may occur to you that what we did in one dimension (vectors) and two dimensions (matrices) could be extended to three, four, or even more dimensions. R can indeed let us define a structure of elements with any number of dimensions: it's called an *array*.

To create one, we use the *array*() function, and specify the dimensions with the *dim* parameter. *dim* is *a vector of the sizes for each dimension of the array.* For instance, suppose we wanted to model the air pressure at various places in a wind tunnel. This chamber is clearly three-dimensional, so we need a 3D array, with coordinates for length (*x* axis), width (*y* axis), and height (*z* axis). Let's say our chamber is 5 meters long, 5 meters wide, and 3 meters high, and we want to measure the air pressure at one-meter intervals. We can create an array to hold the data like this:

```
> airPressure = array(dim=c(5,5,3))
```

Perhaps at the beginning of our experiment, the air pressure is consistent across the length and width of the chamber, and it is slightly higher at lower heights. We'll say that

it's 1 bar (a "bar" is a standard measure of atmospheric pressure) near the bottom of the chamber, .99 bars in the middle, and .98 bars at the top. Here's how we could initialize our three-dimensional array:

```
for (i in 1:5) {
    for (j in 1:5) {
        airPressure[i,j,1] = 1
        airPressure[i,j,2] = .99
        airPressure[i,j,3] = .98
    }
}
> airPressure
, , 1
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
[2,]    1    1    1    1    1
[3,]    1    1    1    1    1
[4,]    1    1    1    1    1
[5,]    1    1    1    1    1
, , 2
     [,1] [,2] [,3] [,4] [,5]
[1,] 0.99 0.99 0.99 0.99 0.99
[2,] 0.99 0.99 0.99 0.99 0.99
[3,] 0.99 0.99 0.99 0.99 0.99
[4,] 0.99 0.99 0.99 0.99 0.99
[5,] 0.99 0.99 0.99 0.99 0.99
, , 3
     [,1] [,2] [,3] [,4] [,5]
[1,] 0.98 0.98 0.98 0.98 0.98
[2,] 0.98 0.98 0.98 0.98 0.98
[3,] 0.98 0.98 0.98 0.98 0.98
[4,] 0.98 0.98 0.98 0.98 0.98
[5,] 0.98 0.98 0.98 0.98 0.98
```

The output looks a little strange at first, until you recognize the pattern. Each cell is identified by three coordinates now, and since the screen is obviously two-dimensional, R lists the contents of each height location separately. The first 5 × 5 group of cells is labeled "`, , 1`" at the top, which means "here are all the rows and columns for height value 1." The resulting group can be read just like a (2-dimensional) matrix, for that is what it is. Then the cells at heights 2 and 3 follow. Notice that inside the body of our nested for loop, we set the values for all three cells at row *i* and column *j* by varying the third index from 1 to 3.

Multidimensional arrays work just the same as matrices: we can extract a cell or group of cells by specifying indices, or vectors of indices. If we wanted columns 4 and 1 (in that order) of rows 2 through 5 for all the heights, we would write:

```
airPressure[c(4,1),2:5,]
```

leaving the third index blank since we want all the heights.

Quick Review Question 3 Given the *airPressure* matrix, as defined above, write *R* expressions to return each of the following:

a. The element at row 3, column 3 at height 3.

b. The element at row 4, column 2 at height 1.

c. The entire 3rd row for all columns and heights.

d. All rows and columns for the lowest height.

e. All heights for row 4, columns 2 through 5.

From Tutorial 3

2 Vector operations

Vectors are essential to R, and we can perform operations on entire vectors. To perform addition, subtraction, multiplication, division, or exponentiation of a scalar (number) by every element in a vector, we use the usual operator of +, -, *, /, or ^, respectively. For example, consider the following vector:

```
> vec = seq(4, 5.2, .3)
> vec
[1] 4.0 4.3 4.6 4.9 5.2
```

The following input statements (in red) perform the same operation on every element of vec, returning the appropriate output array (in blue) without changing the value of vec:

```
> vec + 10
[1] 14.0 14.3 14.6 14.9 15.2
> 3 + vec
[1] 7.0 7.3 7.6 7.9 8.2
> vec - 5
[1] -1.0 -0.7 -0.4 -0.1 0.2
> vec * 10
[1] 40 43 46 49 52
> vec / 10
[1] 0.40 0.43 0.46 0.49 0.52
> vec ^ 3
[1] 64.000 79.507 97.336 117.649 140.608
> 2 ^ vec
[1] 16.00000 19.69831 24.25147 29.85706 36.75835
```

To change the value of vec, we must assign the result of an operation to vec, such as follows:

```
> vec = vec * 10
> vec
[1] 40 43 46 49 52
```

These techniques apply not only to 1-dimensional vectors, but also to multi-dimensional matrices and arrays. For example:

```
> mat = matrix(c(3,5,2,4,4,1),nrow=3)
```

```
> mat
     [,1] [,2]
[1,]    3    4
[2,]    5    4
[3,]    2    1
> mat + 7
     [,1] [,2]
[1,]   10   11
[2,]   12   11
[3,]    9    8
```

As the following Quick Review Question illustrates, we can also apply other functions, such as the square root function (*sqrt*()), that operate on a single number to every element of a vector.

Quick Review Question 1

a. With one assignment statement, make *qrq* be a 2-by-4 matrix of all zeros.

b. With one assignment statement, make the first row of *qrq* be the sequence of positive integers 1, 3, 5, 7.

c. Return the product of 3 by every element of *qrq* without changing *qrq*.

d. Return the square root of every element of *qrq* without changing *qrq*.

e. Add 2 to every element of *qrq*, changing the value of *qrq* to hold those increased numbers.

If two arrays[2] have the same dimensions, we can perform operations that combine corresponding elements, using the typical operators +, -, *, /, and ^. If you've had linear algebra before, **note that the * operator performs an element-by-element multiplication, *not* matrix multiplication.** The matrix multiplication operator is **%*%** (all three symbols typed in a row).

The following statements illustrate these operations using *vec* and another five-element vector:

```
> vec = c(40,43,46,49,52)
> vec2 = c(3,-1,0,8,1)
> vec * vec2
[1] 120 -43 0 392 52
> vec2 / vec
[1] 0.07500000 -0.02325581 0.00000000 0.16326531 0.01923077
> vec ^ vec2
[1] 6.400000e+04 2.325581e-02 1.000000e+00 3.323293e+13 5.200000e+01
```

The last result includes entries in scientific notation, since the numbers were so large (49^8 is a large number indeed). When R prints a number like "5.3e+6", this means "5.3×10^6". (The "e" stands for "exponent" and does *not* indicate the constant $e \approx$

2 Or vectors (recall that a vector is just a one-dimensional array).

2.71828.) So the fourth entry in the above vector is the value 3.323293×10^{13}.

Many functions in R can combine the values in a vector. For example, the function *sum*() returns the sum of all the elements in a vector:

```
> vec = c(40,43,46,49,52)
> sum(vec)
[1] 230
```

Other similar functions include *max*(), *min*(), *prod*(), *mean*(), and *sd*() (for "standard deviation"). Experiment with these to see their effects. All of these functions work on multi-dimensional arrays as well as vectors, too:

```
> mat = matrix(c(5,3,8,4,3,6),nrow=3)
> mat
     [,1] [,2]
[1,]    5    4
[2,]    3    3
[3,]    8    6
> sum(mat)
[1] 29
> max(mat)
[1] 8
> mean(mat)
[1] 4.833333
```

Combining and Transposing Arrays

Sometimes we have separate vectors of numbers that we need to combine. An example would be if we had a vector of *x*-coordinates and a vector of *y*-coordinates, and we wanted to combine them to get a matrix of ordered pairs, where each row represented one ordered pair (*x*,*y*). Suppose for an hour a scientist measures amounts (in milligrams) of residues from a chemical reaction every 12 minutes, or 0.2 hours. The following command assigns to tlst the list of times:

```
> tlst = seq(0,1,.2)
> tlst
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

The following rlst is a list of residue measurements:

```
> rlst = c(0,.05,.16,.23,.55,1)
```

We can use the *cbind*() function (for "column bind") to bind these two vectors togethers as columns of a matrix:

```
> combinedlst = cbind(tlst,rlst)
> combinedlst
     tlst rlst
[1,]  0.0 0.00
[2,]  0.2 0.05
[3,]  0.4 0.16
[4,]  0.6 0.23
[5,]  0.8 0.55
[6,]  1.0 1.00
```

We now have one time/residue ordered pair per row.

Another useful operation is to take the **transpose** of a matrix. A matrix's transpose is just the same matrix with the rows and columns interchanged: what were the rows become the columns, and vice versa. In R, the *t()* function performs the **transpose**:

```
> t(combinedlst)
      [,1] [,2] [,3] [,4] [,5] [,6]
tlst     0 0.20 0.40 0.60 0.80    1
rlst     0 0.05 0.16 0.23 0.55    1
```

Note that R preserves the names of the original rows, which is handy for interpreting data once you've combined it.

Quick Review Question 3 Write a statement to generate a list xlst of x-values, which are positive integers from 1 through 9. Using one assignment statement, have *glst* store the corresponding values of $3\sqrt{x}$. Write commands to assign to pairslst the array of ordered pairs with one ordered pair per row.

From Tutorial 4

- **Random Numbers**

Random numbers are essential for computer simulations of real-life events, such as weather or nuclear reactions. To pick the next weather or nuclear event, the computer generates a sequence of numbers, called **random numbers** or **pseudorandom numbers**. As we discuss in Module 9.2, "Simulations," an algorithm actually produces the numbers; so they are not really random, but they appear to be random. A uniform random number generator produces numbers in a **uniform distribution** with each number having an equal likelihood of being anywhere within a specified range. For example, suppose we wish to generate a sequence of uniformly distributed, four-digit random integers. The algorithm used to accomplish this should, in the long run, produce approximately as many numbers between, say, 1000 and 2000 as it does between 8000 and 9000.

Definition **Pseudorandom numbers** (also called **random numbers**) are a sequence of numbers that an algorithm produces but which appear to be generated randomly. The sequence of random numbers is **uniformly distributed** if each random number has an equal likelihood of being anywhere within a specified range.

R provides the function *runif()* to generate uniform random numbers. The simplest way to use it is to give it a single argument, specifying how many random numbers you want. Each call to *runif* returns a vector of uniformly distributed pseudorandom floating point numbers between 0 and 1. Evaluate the following commands several times to observe the generation of different random numbers:

```
runif(1)
runif(5)
runif(10)
```

Suppose, however, we need our random floating point numbers to be in the range from 2.0 up to 5.0. We can specify *min* and *max* parameters to *runif()* to get a different range:

```
runif(10, min=2, max=5)
```

We can use the *matrix*() command, providing a random vector and a number of columns (or rows) to create a two-dimensional random matrix:

```
matrix(runif(12, min=0, max=10), ncol=3)
```

Quick Review Question 1

a. Generate a single random floating point number between 10 and 100.
b. Generate a length-5 vector of random numbers between -3 and 3.
c. Generate a 2-by-4 matrix of random numbers between 8 and 12.

From Tutorial 5

Function Files

Previously, we have used anonymous functions with very short definitions and, consequently, have defined them in one script file with the entire program. For example, we can define a function, *sqr*, as follows:

```
sqr = function(x) {x*x}
```

However, for a function that has a longer definition or that we wish to reuse, we place the definition in a separate file. To begin we select *New Document* from the *File* menu. In the resulting file, we type the function with appropriate comments, such as follows:

```
# function to return square of a parameter
sqr = function(x) {x*x}
```

R returns the last value in the function definition, here *x*x*. Alternatively, we explicitly use a ***return***, as follows:

```
# function to return square of a parameter
sqr = function(x) {
   return(x*x)
}
```

After writing this function file, we save the file using the name of the function, here *sqr*, and *R* appends the extension ***.R***. Thus, the file name is *sqr.R*. For *R* to accept input from this file after initially defining or after making any change, we must execute the *source* command from the command line of the main script file, as follows:

```
source("sqr")
```

A simulation often includes a number of such function files. Thus, we can organize all the *source* commands in one file, say *source.R*, that also removes all earlier definitions, as follows:

```
# File:  source.R
rm(list=ls(all=TRUE))
source("sqr.R")
… # other source commands appear here
```

Then, the main script can execute one *source* command for this file, resulting in execution of all *source* commands:

```
# main script file
source("source.R")
```

Once we save the function file and execute the appropriate *source* command, from the command window or in the program, we can call the function with an argument of 4, as follows:

```
sqr(4)
```

We should get the answer of 16. However, we may get a message, such as follows:

```
Error: could not find function "sqr"
```

In this case, we need to inform *R* where to look for the definition. From the *Misc* menu, we click *Change Working Directory…*, browse to the appropriate folder, and click *Open*. When we call a function, *R* searches the saved path names until finding a match.

Quick Review Question 4

a. Define a function, *rectCircumference*, that returns the circumference, *circumference*, of a rectangle with parameters for length and width, l and w, respectively. The circumference is $2l + 2w$. Use a function file. If necessary, set the path to the function definition.

b. In the answer script file, have the appropriate *source* command. Call the function to return the circumference of a rectangle with dimensions 3 and 4.2, respectively.

If we have several values to return, we can place them in a vector. For example, the following line begins a function definition in which the function returns the area and circumference of a circle with radius *r:*

```
circleStats = function(r) {
```

In the function body, we have two lines for the area and circumference and return a vector of these values, as follows:

```
area = pi * r * r
circumference = 2 * pi * r
return(c(area, circumference))
```

We employ the operator .* so that *circleStats* can operate on a vector of radii, such as *circleStats*([1 3]).

In calling the function, we assign the function call to a variable, such as follows:

```
stats = circleStats(5)
```

Execution of the command assigns the area and circumference of the circle with radius 5 to vector *stats*, as the following shows

```
> stats
[1] 78.53982 31.41593
```

Referencing individual elements, we can store the values in separate variables, as follows:

```
ar = stats[1]
cir = stats[2]
```

Quick Review Question 6

a. Define a function *squareStats* that returns the area (*side* squared) and circumference (four times *side*) of a square with a parameter, *side*, for the length of a side. Use a function file.
b. Execute an appropriate *source* command.
c. Call the function to obtain the area and circumference of a square with each side having length 3.
d. Assign the area and circumference to individual variables, *area* and *circumference*.

While Loop

The material in this section is useful for Chapter 13 and several projects in Chapter 14 that are appropriate after covering the current chapter.

We have employed the *for* loop to repeat a segment of code when we know the number of iterations. However, if a loop must execute as long as a condition is true, we can use a ***while*** loop. The form of the command is as follows:

```
while (condition) {
    statements
}
```

For example, the segment below generates and displays random numbers between 0.0 and 1.0 as long as the values are less than 0.7. The segment also counts how many of the random values are in that range.

```
counter = 0
ra = runif(1)
while (ra < 0.7) {
    counter = counter + 1
    ra = runif(1)
}
counter
```

We initialize to zero a variable, *counter*, that is to count the number of random numbers less than 0.7. Before the loop begins, we prime *ra* with a random number so that *ra* has an initial value to compare with 0.7. Then, at the end of the loop, we obtain and display another value for *ra* to compare with 0.7. After completion of the loop, we display the final value of *counter*.

Quick Review Question 1 Write a segment to generate an animation, as follows: Assign 0 to *x* and 1 to *i*, an index. While *x* is between -5 and 5, plot the point (*x*, 0) as a small circle; save the frame as the *i*th element of a vector; add 1 to *i*; and use *randIntRange* from Quick Review Question 5 to generate a random integer -1, 0, or 1 and assign to *x* the sum of this number and *x*.

From Tutorial 7

Matrix Operations for Modules 13.2-13.4

When two matrices have the same size, we can perform the **+** or * operation. The resulting matrix has the same size and each element of the result is the sum or product, respectively, of corresponding elements of the two operands. The following commands record the elements of matrices one row at a time using, ***byrow = TRUE***, and give examples of these operators:

```
matA = matrix(c(3, 6, 5, -2, 0, 3), nrow = 2, byrow = TRUE)
matB = matrix(c(4, -1, 0, 7, 8, 3), nrow = 2, byrow = TRUE)
matA + matB
matA * matB
```

As the results show, adding element by element, the sum *matA* + *matB* = $\begin{bmatrix} 3 & 6 & 5 \\ -2 & 0 & 3 \end{bmatrix}$ + $\begin{bmatrix} 4 & -1 & 0 \\ 7 & 8 & 3 \end{bmatrix}$ is $\begin{bmatrix} 7 & 5 & 5 \\ 5 & 8 & 6 \end{bmatrix}$. Similarly, multiplying element-by-element for *matA* .* *matB*, we obtain $\begin{bmatrix} 12 & -6 & 0 \\ -14 & 0 & 9 \end{bmatrix}$.

For vectors that have the same number of elements, we can perform the **dot product**, which returns a number, the sum of the product of corresponding elements. The *R* function ***sum*** performs the operation on the elements of a vector, which is the element-by-element product of two vectors, accomplished with *. Thus, the following command returns the dot product of [2 7 -1] and [5 3 4], which is $2 \cdot 5 + 7 \cdot 3 + -1 \cdot 4 = 10 + 21 - 4 = 27$:

```
sum(c(2, 7, -1) * c(5, 3, 4))
```

The matrix-times-vector or matrix product operator is *. Consider vector *vecCol* = $\begin{bmatrix} 7 \\ 1 \\ -2 \end{bmatrix}$, which the following command creates:

```
vecCol = c(7, 1, -2)
```

The product of *matA* and *vecCol*, *matA* ***%*%*** *vecCol*, produces a 2-by-1 column vector, $\begin{bmatrix} 17 \\ -20 \end{bmatrix}$. With *matC* = $\begin{bmatrix} 7 & -3 \\ 1 & 0 \\ -2 & 6 \end{bmatrix}$, the following is an example of matrix multiplication, producing the result $\begin{bmatrix} 27 & 42 & 26 \\ 3 & 6 & 5 \\ -18 & -12 & 8 \end{bmatrix}$:

```
matC = matrix(c(7, -3, 1, 0, -2, 6), nrow = 3, byrow = TRUE)
matC %*% matA
```

Quick Review Question 1 Start a new *Script*. In opening comments, have "R Tutorial 7 Answers" and your name. Save the file under the name *RCTTutorial7Ans.R*. In

the file, preface this and all subsequent Quick Review Questions with a comment that has "## QRQ" and the question number, such as follows:

```
## QRQ 1 a
```

a. Generate a 4-by-2 matrix *mA*, where the *i*-*j* element is the sum of *i* and *j*. Thus, after forming *mA*, *mA*[3, 2] should be 5, which is 3 + 2.
b. Generate a 4-by-2 matrix *mO* of all ones.
c. Give matrix sum of *mA* and *mO*.
d. Define a vector *u* with elements 2 and 7.
e. Define a vector *v* with elements 5 and 3.
f. Give dot product of *u* and *v*.
g. Generate a 2-by-3 matrix *mB*, where the *i*-*j* element is the difference of *i* and *j*, *i* - *j*.
h. Give the matrix product of *mA* and *mB*.

We can take the matrix product of a square matrix by itself, which accomplishes the matrix power operation. For example, suppose *mS* is defined as follows:

```
mS = matrix(c(8, 4, 6, 7, 7, 1, 4, 6, 2), nrow = 3, byrow = TRUE)
```

The matrix product of *mS* with itself, *mS* %*% *mS*, is the 3 × 3 matrix $\begin{bmatrix} 116 & 96 & 64 \\ 109 & 83 & 51 \\ 82 & 70 & 34 \end{bmatrix}$. Mathematically, we can generate the same result by squaring *mS*, mS^2. In *R*, for larger powers, such as 5, we can start by assigning the identity matrix, ***diag*(3)**,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

to a variable, such as *prod*, and with a *for* loop repeatedly multiply by *mS*, as follows:

```
prod = diag(3)
for (i in 1:5) {
    prod = prod %*% mS
}
```

Quick Review Question 2

a. Generate a 3 × 3 matrix *mC*, where the *i*-*j* element is 2*i* - *j*.
b. Calculate the matrix power mC^7 using a loop as above.
c. Show that the calculation of the matrix product of *mC* with itself seven times has the same value as the answer to Part b.

To test if all elements of a matrix or vector satisfy a condition, we employ the command ***all***. Similarly, to test if any of these elements satisfy a condition, we use ***any***. Thus, the answer to the following segment is *TRUE*:

```
b = c(3, 5, 2)
all(b == b)
```

Moreover, because 5 > 4, the following command returns *TRUE*:

```
any(b > 4)
```

Eigenvalues and Eigenvectors for Modules 13.3-13.4

For square matrix *M*, the constant λ is an **eigenvalue** and *v* is an **eigenvector** if multiplication of the constant by the vector accomplishes the same results as multiplying the matrix by the vector, that is, the following equality holds:

$Mv = \lambda v$

The **dominant eigenvalue** for a matrix is the largest eigenvalue for that matrix. The *R* function ***eigen*** returns a vector containing the eigenvalues and a matrix with the associated eigenvectors for a square matrix argument, as the following illustrates:

```
mat = matrix(c(0, 3, 6, 0.1, 0, 0, 0, 0.4, 0), nrow=3, byrow=TRUE)
eigen(mat)

$values
[1]  0.7796379+0.0000000i -0.3898189+0.3948119i -0.3898189-0.3948119i

$vectors
                [,1]                   [,2]                   [,3]
[1,] 0.98976800+0i -0.9761933+0.0000000i -0.9761933+0.0000000i
[2,] 0.12695227+0i  0.1236176+0.1252010i  0.1236176-0.1252010i
[3,] 0.06513397+0i  0.0016143-0.1268359i  0.0016143+0.1268359i
```

In this case, two of the eigenvalues are complex numbers, and the dominant eigenvalue is the first element of lambda, 0.7796379. (0.0000000i is 0.) The corresponding eigenvectors are in the columns of *$vectors*. Thus, the corresponding eigenvector, (0.98976800, 0.12695227, 0.06513397), of the dominant eigenvalue is the first column of *$vectors*, ignoring 0i.

To place the eigenvectors in separate variables, we store the results in a variable and employ ***$values*** and ***$vectors***, as follows:

```
eig = eigen(mat)
lambdaLst = eig$values
vecLst = eig$vectors
```

The following illustrates that the matrix-vector product of *mat* and this vector equals the dominant eigenvalue times the vector.

```
lambda = lambdaLst[1]
v = vecLst[,1]
mat %*% v
lambda * v
```

Each of the products returns the vector $\begin{bmatrix} 0.7717 \\ 0.0990 \\ 0.0508 \end{bmatrix}$.

Quick Review Question 3

a. Define the matrix *mD* as $\begin{bmatrix} 4 & 6 \\ 3 & 1 \end{bmatrix}$.

b. Return a list of eigenvalues of *mD* without calculating eigenvectors.

c. Calculate a list of eigenvalues and eigenvectors of *mD*, and store the results in variables *lLst* and *vLst*, respectively.
d. Verify that the matrix-vector product of *mD* and the dominant eigenvalue equals the product of that eigenvalue and the corresponding eigenvector.

Additional Commands Used in Module 13.5

In this section, we consider several *R* commands that are used in the file associated with Module 13.5.

As execution of the following shows, the *R* command ***unique*** with a list argument returns a list with the same elements but without duplicates, in this case (1, 5, 4). The function does not change the argument, *lst*.

```
lst = c(1, 5, 5, 4, 1, 5)
unique(lst)
lst
```

The function ***union*** returns the sorted union of two list arguments with no duplicates. Thus, output of the following is the list 1 5 4 8 7 3:

```
lst1 = c(1, 5, 5, 4, 1, 5)
lst2 = c(4, 8, 7, 8, 3)
union(lst1, lst2)
```

Using the command ***order***, we can sort rows of a matrix in ascending or descending order based on a particular column. Consider the matrix $\begin{bmatrix} 5 & 4 & 5 \\ 1 & 7 & 8 \\ 2 & 3 & 9 \\ 3 & 5 & 6 \end{bmatrix}$, which we define as follows:

```
triples = matrix(c(5, 4, 5, 1, 7, 8, 2, 3, 9, 3, 5, 6),
                 nrow = 4, byrow = TRUE)
```

To sort in ascending order, based on the values in the second column, in brackets after the matrix name, we use *order* with an argument of the second column and a comma, as follows:

```
triples[order(triples[,2]), ]
```

The sorted output, with the second column elements in ascending order, is as follows:

```
     [,1] [,2] [,3]
[1,]    2    3    9
[2,]    5    4    5
[3,]    3    5    6
[4,]    1    7    8
```

With a negative in front of the argument for *order*, the sort is in descending order. Consider the following command:

```
triples[order(-triples[,2]), ]
```

The results reveal a sorting on the second column in descending (reverse) order:

```
ans =

     1     7     8
     3     5     6
     5     4     5
     2     3     9
```

Quick Review Question 4

a. Define a list (matrix), *lst1*, of a five numbers, where each element is a random integer between 0 and 2. Display *lst1*.

b. Using *unique* and an assignment statement, eliminate the duplicate pairs in *lst1*. Display *lst1*after the *unique* command.

c. Form another list, *lst2*, of 5 numbers, where each element is a random integer between 10 and 12. Assign the union of *lst1* and *lst2* to *lst3*. Display *lst2* and *lst3*.

d. Define a list, *dup1*, of a five ordered pairs of numbers, where the first element in each pair is a random integer between 0 and 2 and the second element is a random integer between 10 and 12. Write a command to return *dup1*, sorted by the first elements from highest to lowest.