9.4 *R* Tutorial 5

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.

The prerequisites to this tutorial are *RCT* Tutorials 1-4. Tutorial 5 prepares you to use *R* for material in this and subsequent chapters. The tutorial introduces the following functions and concepts: maximum, minimum, length, animation, function files, and several features for projects, including the *while* loop, logical operators, and testing membership.

Maximum, Minimum, and Length

The *R* function ***max*** returns the maximum of a vector argument. Thus, the form of a call to the function can be as follows:

```
max(c(x1, x2, ... ))
```

In this case, the function returns the numerically largest of *x1*, *x2*, ... For example, the following invocation with four arguments returns the maximum argument, 7.

```
max(c(2, -4, 7, 3))
```

Another form of a call to the function, which follows, has a two-dimensional array of numbers as argument. The result is a vector of the maximum of all the elements in the matrix. The segment below generates a 3-by-3 array of numbers and returns the maximum value 5:

```
vec = c(-1, 2, -1, 0, 1, 5, 3, 2, 1)
lst = matrix(vec, nrow=3)
max(lst)
```

In a similar fashion, the *R* function ***min*** returns the minimum of a vector of numeric arguments; or in the case of a two-dimensional array, *min* returns a vector a minimum column values.

When we do not know the length of a vector, we can employ the ***length*** function, such as follows:

```
length(lst)
```

If *lst* were a matrix, this command would return the number of its elements.

Quick Review Question 1 Start a new *Script*. In opening comments, have "R Tutorial 5 Answers" and your name. Save the file under the name *RCTTutorial5Ans.m*. 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. Write a segment to generate a list, *lst*, of 100 random floating point numbers between 0 and 1 and to display the maximum and minimum in the list.

b. Write a segment to generate and display a vector of zeros, *z*, where the number of elements is a random integer between 1 and 20, and to return the number of elements in *z*.

Animation

One interesting use of a *for* loop is to draw several plots for an animation. In making the graphics, plots should have the same *axis* so that axes appear fixed during the animation. The segment below generates a sequence of plots of 0.5 sin(*x*), sin(*x*), 1.5 sin(*x*), ..., 5.0sin(*x*) with *x* varying from 0 to 2π to illustrate the impact the coefficient has on amplitude. However, for an effective demonstration, each graph must have the same display area, here from -5 to 5 on the *y*-axis.

```
x = seq(0, 2*pi, 0.1)
for (i in 1:10) {
   y = 0.5 * i * sin(x)
   plot(x, y, type = 'l', xlim=c(0,2*pi), ylim=c(-5,5))
}
```

However, the animation can be fairly slow and jerky as *R* generates the animation one frame at a time.

To display the animation slower, we can insert a command, ***Sys.sleep(x)***, to pause for a designated number of seconds. For example, *the following command* stops further execution for 5 seconds:

```
Sys.sleep(5)
```

The adjusted sequence to generate an animation with approximately 4 frames per second follows:

```
x = seq(0, 2*pi, 0.1)
for (i in 1:10) {
   y = 0.5 * i * sin(x)
   plot(x, y, type = 'l', xlim=c(0,2*pi), ylim=c(-5,5))
   Sys.sleep(0.25)
}
```

Quick Review Question 2 The first two statements of following segment generate 10 random *x*- and *y*-coordinates between 0 and 1. We then plot the points as fairly large black dots in a 1-by-1 rectangle. Adjust the code to generate 10 plots. The first graphics only displays the first point; the second graphics shows the first two points; and so forth. Why do we want to specify the plot range to be the same for each graphic?

```
xLst = runif(10)
yLst = runif (10)
plot(xLst, yLst, xlim=c(0,1), ylim=c(0,1))
```

Quick Review Question 3 The function *f* below is the logistic function for constrained growth, where the initial population is 20, the carrying capacity is 1000, and the rate of change of the population is $(1 + 0.2i)$ (see Module 2.3, "Constrained Growth"). To see the effect of increasing the rate of change from $(1 + 0.2(\mathbf{1})) = 1.2$ to $(1 + 0.2(\mathbf{10})) = 3.0$, replace the assignment of 5 to *i* with the start of a *for* loop to generate 10 plots of $f(t, i)$, where *i* varies from 1 to 10.

```
f = function(t, i) {(1000*20)/((1000 - 20)*exp(-(1+0.2*i)*t) + 20)}
t = seq(0,3,0.1)
i = 5  # replace with for loop
plot(t, f(t, i) , type = 'l', xlim=c(0,3), ylim=c(0,1000))
```

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.

Quick Review Question 5

a. In a function file, define *randIntRange* with parameters *lower* and *upper* to return a random integer between *lower* and *upper* -1, inclusively. Thus, *randIntRange*(-3, 3) should return a number from the set {-3, -2, -1, 0, 1, 2}.

b. Have an appropriate *source* file.

c. Write a *for* loop to display 10 random integers between 5 and 8, inclusively.

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*.

By default, variables inside the definition are **local** to the function. To make symbols known elsewhere, we declare them to be ***utils::globalVariables***. However, we should use this feature with great care, because global variables can cause unexpected results, called **side effects**. It is appropriate to declare constants, usually written in all uppercase letters, that several functions use as global. If needed, the command window or main script file should also declare these constants as *globalVariables* in a vector. The following declares *EMPTY* to be a global variable and assigns it the value 0.

```
utils::globalVariables("EMPTY")
EMPTY = 0
```

For more than one global variable, we employ a vector, as follows:

```
utils::globalVariables(c("EMPTY", "TREE"))
```

Logical Operators

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

Sometimes, a condition, such as in an *if* statement, is **compound**. For example, suppose we wish to display "Out of bounds" if x is less than -3 or greater than 3. A **logical OR operator** in *R* is ||, so the statement is as follows:

```
if ((x < -3) || (x > 3)) {
    cat("Out of bounds\n")
}
```

Although obviously not in this example, sometimes both conditions can be true, such as with $(x > 3) \,||\, (y > 3)$ when x is 5 and y is 10. In this case, the compound condition is also true.

The logical OR operator employs **short circuiting**. With short circuiting, as soon as *R* can detect that a compound condition is true or false, the system stops computation of the condition. In the above example, $((x < -3) \,||\, (x > 3))$, if the value of x is less than -3, we know that the compound condition is true; *R* does not need to evaluate the second condition, $x > 3$.

The opposite condition, $-3 \le x \le 3$, requires a **logical AND operator &&**, as in the following example:

```
if ((-3 <= x) && (x <= 3)) {
    cat("Out of bounds\n")
}
```

We cannot write the condition as in mathematics, $(-3 \le x \le 3)$, but must express the condition with a compound statement. The operator && also employs short circuiting. For example, if x is -5, we know that the condition $(-3 \le x)$ is false; and, in fact, we know the compound condition $((-3 \le x)$ && $(x \le 3))$ is false without calculating the second condition. Thus, because of short circuiting, *R* would not evaluate $(x \le 3)$.

To negate a condition, we employ the **logical NOT operator !**. Thus, $(!(x > 3))$ is equivalent to $(x <= 3)$.

Expressions can involve logical operators in conjunction with arithmetic and relational operators. In such cases, the operator precedence of Table 1 determines the order of evaluation. When in doubt, we can always use parentheses to clarify the precedence as in the above two *if* statements. However, as Table 1 indicates, we can omit the parentheses, as follows, because *R* evaluates an inequality before evaluating an OR operator:

```
if (x < -3 || x > 3) {
    cat("Out of bounds\n")
}
```

Table 1 Operator precedence from highest to lowest

1.	()					
2.	'	^	.'	.^		
3.	Unary +	Unary -	Unary ~			
4.	*	/	.*	./		
5.	+	-				
6.	:					
7.	<	<=	>	>=	==	~=
8.	&&					
9.	||					

Quick Review Question 7 Write an *if* statement for the following situation and test using several values for the variables: If $x + 2$ is greater than 3 or y is less than x, add 1 to y; otherwise, subtract 1 from x.

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 8 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 ith 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.