9.1 R Tutorial 4

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. See the R computational toolbox tutorial on the book's website for instructions on downloading and installing R.

The prerequisites to this tutorial are R Computational Toolbox Tutorials 1-3. Tutorial 4 prepares you to use R for material in this and subsequent chapters. The tutorial introduces the following functions and concepts: random numbers, modular arithmetic, if statement, count, standard deviation, and histogram.

Besides being a system with powerful commands, R is a programming language. In this tutorial, we consider some of the commands for the higher level constructs (selection and looping) that are available in R. Besides more traditional applications, looping enables us to generate graphical animations that help in visualization of concepts. The statistical functions are of particular interest in Module 9.3 on "Area through Monte Carlo Simulation." That module also employs several additional R commands, which this tutorial introduces.

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 Start a new *Script*. In opening comments, have "R Tutorial 4 Answers" and your name. Save the file under the name *RCTTutorial4Ans.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 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.

To obtain an integer random number, we use the ***floor***() function in conjunction with *runif*(). *floor*() returns the integer immediately below the floating-point number it is passed as an argument; for instance *floor*(6.8) returns the answer 6. Thus, the following command returns a random integer from the set {1, 2, …, 25}:

```
floor(runif(1, min=1, max=26))
```

Note that we had to make the *max* 26 instead of 25, since all floating-point numbers in the range 25.0 to 26.0 will get "floored" to the integer value 25.

Combining these operations, the following commands assign to *matSqr* a 4-by-4 array of integers from the set {11, 12, …, 19} and to *matRect* a 2-by-3 array of integers from the same set:

```
matSqr = matrix(floor(runif(16, min=11, max=20)), ncol=4)

matRect = matrix(floor(runif(6, min=11, max=20)), ncol=3)
```

Quick Review Question 2

a. Give the command to generate a number representing a random throw of a die with a return value of 1, 2, 3, 4, 5, or 6.
b. Give the command to generate 20 random numbers representing ages from 18 to 22, inclusively, that is, random integers from the set {18, 19, 20, 21, 22}.

A random number generator starts with a number, which we call a **seed** because all subsequent random numbers sprout from it. The generator uses the seed in a computation to produce a pseudorandom number. The algorithm employs that value as the seed in the computation of the next random number, and so on.

Typically, we seed the random number generator once at the beginning of a program, using the function ***set.seed***(). For example, we seed the random number generator with 14234 as follows:

```
set.seed(14234)
```

If the random number generator always starts with the same seed, it always produces the same sequence of numbers. A program using this generator performs the same steps with each execution. The ability to reproduce detected errors is useful when debugging a program.

However, this replication is not desirable when we are using the program. Once we have debugged a function that incorporates a random number generator, such as for a computer simulation, we want to generate different sequences each time we call the function. For example, if we have a computer simulation of weather, we do not want the program always to start with a thunderstorm. By default, if the seed is not set, R initializes it to something different every time you run your program.

Quick Review Question 3

a. Write a command to generate a 10-by-10 matrix of random integers from 1 through 100, inclusively, that is, from the set {1, 2, 3, ..., 99, 100}. Using the up-arrow to repeat the command, execute the expression several times, and notice that the list changes each time.

b. Using the up-arrow, retrieve the command from Part a. Before the command, seed the random number generator with a four-digit number. Execute the two commands in sequence several times, and notice that the list does not change.

Modulus Function

An algorithm for a random number generator often employs the modulus operator, **%%** in R, which gives the remainder of a first integer argument divided by a second integer argument. To return the remainder of the division of m by *n*, we employ a command of the following form:

```
m %% n
```

(This call is equivalent to *m* % *n* in C, C++, and Java). Thus, the following statement returns 3, the remainder of 23 divided by 4.

```
23 %% 4
```

Quick Review Question 4 Assign 10 to *r*. Then, assign to *r* the remainder of the division of 7*r* by 11. Before executing the command, calculate the final value of *r* to check your work.

Selection

The **flow of control** of a program is the order in which the computer executes statements. Much of the time, the flow of control is sequential, the computer executing statements one after another in sequence. We refer to such segments of code as a **sequential control structure**. A **control structure** consists of statements that determine the flow of control of a program or algorithm. The **looping control structure** enables the computer to execute a segment of code several times. In the first R tutorial, we considered the function *for*, which is one implementation of such a structure.

Definitions The **flow of control** of a program is the order in which the computer executes statements. A **control structure** consists of statements that determine the flow of control of a program or an algorithm. With a **sequential control structure**, the computer executes statements one after another in sequence. The **looping control structure** enables the computer to execute a segment of code several times.

A **selection control structure** can also alter the flow of control. With such a control structure, the computer makes a decision by evaluating a logical expression. Depending on the outcome of the decision, program execution continues in one direction or another.

Definition With a **selection control structure**, the computer decides which statement to execute next depending on the value of a logical expression.

R can implement the selection control structure with an ***if*** statement. One form of the function is as follows:

```
if (condition){
    tStatements
}
```

If *condition* has the value ***true***, then the function executes the statement(s) *tStatements*. For example, the following statement assures that we do not divide by 0:

```
if (count != 0) {
   total/count
}
```

The statements above use the relational operator that indicates **not equal** (!**=**). A **relational operator** is a symbol that we use to test the relationship between two expressions, such as two variables. A common error in R is to forget that the *if* statement's test for equality (**==**) requires two consecutive equal signs, rather than the single one for assignments. Table 1 lists all the relational operators.

Table 1 Relational operators

Relational Operator	Meaning
==	equal to
>	greater than
<	less than
!=	not equal to
>=	greater than or equal to
<=	less than or equal to

Another form of the *if* statement is as follows:

```
if (condition){
    tStatements
} else {
    fStatements
}
```

(Note that unlike in some languages, the curly brace that closes the *tStatements* must be on the same line as the word else.) This form of the statement presents an alternative should *condition* be ***false***. In this case, the function executes the statement(s) *fStatements*. Before executing the following commands, predict the output and the value of *minxy*:

```
x = 3;
y = 5;
if (x < y) {
   minxy = x
} else {
   minxy = y
}
```

Because *x* is less than *y*, the *if* statement assigns *x* (3) to *minxy*. The *if* statement accomplishes the same things as the following pseudocode:

if *x* is less than *y* then
    minxy is assigned *x*
else
    minxy is assigned *y*

Thus, the *if* command returns the smaller of the two values, *x* or *y*.

Quick Review Question 5 Write an *if* statement to generate and test a random floating point number between 0 and 1. If the number is less than 0.3, return 1; otherwise, return 0.

We can "nest" if/else statements to allow for more than two alternatives. The following two forms enable us to test for three choices:

```
if (condition1) {
    body1Statements
} else {
    if (condition2) {
        body2Statements
    } else {
        body3Statements
    }
}
```

Notice that the curly braces "line up": each left curly matches a closing right curly, immediately beneath it. It's an excellent idea to keep your indentation straight like this so you don't lose track of which statements are in which blocks.

Quick Review Question 6 Assign a random floating point number between 0 and 1 to variable *r*. Write a statement to display "Low" if *r* is less than 0.2, "Medium low" if it is greater than or equal to 0.2 and less than 0.5, and "Medium high" if it is greater than or equal to 0.5 and less than 0.8, "High" if it is greater than or equal to 0.8. Note that for each category after low, we do not need to check the first condition. For example, if *r* is not less than 0.2, it certainly is greater than or equal to 0.2. Thus, to determine if "Medium low" should print, we only need to check if *r* is less than 0.5.

Counting

Frequently, we employ vectors in *R*; and instead of using a loop, we can use the function *sum* to count items in the list that match a pattern. As the segment below illustrates, *sum* provides an alternative to a *for* loop for counting the number of random numbers less than 0.3. First, we generate a table of boolean values (trues and falses), such that if a random number (in the range 0.0 to 1.0) is less than 0.3, the table entry is true. Then, since R considers "true" to be "1" for counting purposes, we count the elements in the table that are true:

```
tbl = runif(10) < .3
sum(tbl)
```

Output of the table and the number of ones in the table might be as follows:

```
> tbl
 [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
> 3
```

Quick Review Question 7 Write a statement to generate a table of 20 random floating point numbers between 0 and 1 and store the result in variable *lst*. Give one statement to return the number of these values less than 0.4 and another statement to return the number of values greater than or equal to 0.6.

Quick Review Question 8 Write a segment to generate a vector of 20 random integers between 0 and 5 and return the number of elements equal to 3.

Basic Statistics

The function ***mean*** returns the arithmetic **mean**, or average, of the elements in a vector and has the following format:

```
mean(vector)
```

Similarly, ***sd*** returns the standard deviation of the elements in a vector. The following segment creates a list of 10 floating point numbers in the range 6 to 12 and returns the mean and standard deviation:

```
tbl = runif(10, min=6, max=12)
mean(tbl)
sd(tbl)
```

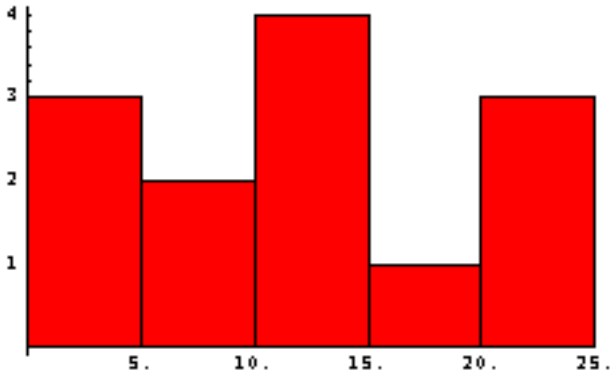
Quick Review Question 9

a. Generate 10,000 normally distributed random numbers using the function ***rnorm*** with the same format as *runif*. Store the results in a vector, *normalNums*. Do not display *normalNums*.
b. Determine the mean of *normalNums*.
c. Determine the standard deviation of *normalNums*.
d. In a normal distribution, the mean is 0 and the standard deviation is 1. Are your answers from Parts b and c approximately equal to these values?

Histogram

A **histogram** of a data set is a bar chart where the base of each bar is an interval of data values and the height of this bar is the number of data values in that interval. For example, Figure 1 is a histogram of the array *lst* = [1, 15, 20, 1, 3, 11, 6, 5, 10, 13, 20, 14, 24]. In the figure, the 13 data values are split into five intervals or categories: [0, 5), [5, 10), [10, 15), [15, 20), [20, 25). The interval notation, such as [10, 15), is the set of numbers between the endpoints 10 and 15, including the value with the square bracket (10) but not the value with the parenthesis (15). Thus, for a number x in [10, 15), we have $10 \le x < 15$. Because four data values (10, 11, 13, 14) appear in this interval, the height of that bar is 4.

Figure 1 Histogram for [1, 15, 20, 1, 3, 11, 6, 5, 10, 13, 20, 14, 24]



Definition A **histogram** of a data set is a bar chart where the base of each bar is an interval of data values and the height of this bar is the number of data values in that interval.

The R command hist() produces a histogram of a list of numbers. The following code assigns a value to *lst* and displays its histogram with a default of 10 intervals or categories:

```
lst = c(1, 15, 20, 1, 3, 11, 6, 5, 10, 13, 20, 14, 24)
hist(lst)
```

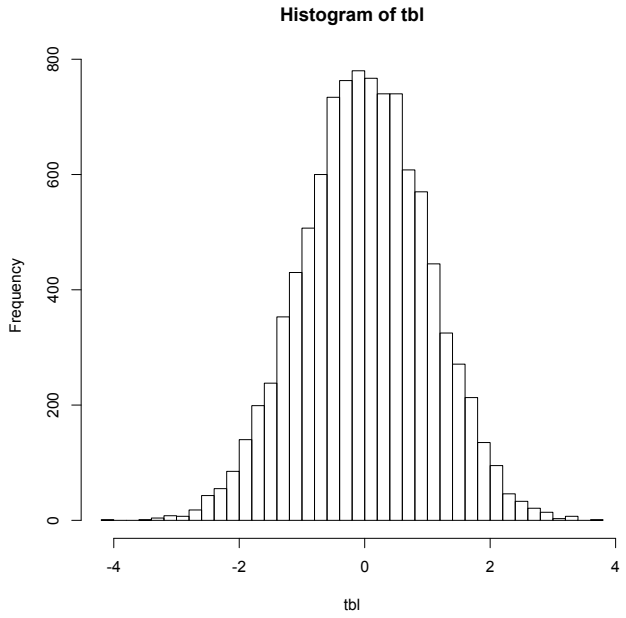
An optional argument called ***breaks*** can be used to specify the number of categories. Thus, the following command produces a histogram similar to that in Figure 1 with five intervals:

```
hist(lst, breaks=5)
```

Figure 2 displays a histogram of an array, *tbl*, of 10000 *normally* distributed random values. The commands to generate the table and produce a histogram with 30 categories are as follows:

```
tbl = rnorm(10000)
hist(tbl, breaks=30)
```

Figure 2 Histogram with 30 categories



Quick Review Question 10

- **a.** Generate a table, *sinTbl*, of 1000 values of the sine of a random floating point number between 0 and π.
- **b.** Display a histogram of *sinTbl*.
- **c.** Display a histogram of *sinTbl* with 10 categories.
- **d.** Give the interval for the last category.
- **e.** Approximate the number of values in this category.