9.1 *MATLAB* Tutorial 4

Introduction to Computational Science: Modeling and Simulation for the Sciences

Angela B. Shiflet and George W. Shiflet
Wofford College
© 2006 by Princeton University Press

Introduction

We recommend that you work through the introduction with a copy of *MATLAB*, answering all Quick Review Questions. *MATLAB* is available from MathWorks, Inc. (http://www.mathworks.com/).

The prerequisites to this tutorial are *MATLAB* Tutorials 1-3. Tutorial 4 prepares you to use *MATLAB* 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, *MATLAB* 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 *MATLAB*. Besides more traditional applications, looping enables us to generate graphical animations that help in visualization of concepts. The statistical toolbox is of particular interest in Module 9.3 on "Area through Monte Carlo Simulation." That module also employs several additional *MATLAB* 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 on "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.

MATLAB provides the random number generator ***rand***. Each call to *rand* returns a uniformly distributed pseudorandom floating point number between 0 and 1. Evaluate

the following command several times to observe the generation of different random numbers:

```
rand
```

For example, executing the command three times generates floating point output between 0 and 1, such as 0.296615, 0.908778, and 0.056155.

With one integer argument, ***rand*(*n*)** returns a square array with *n* rows and *n* columns of random numbers between 0 and 1. Similarly, ***rand*(*m*, *n*)** returns an array with *m* rows and *n* columns of random numbers.

Suppose, however, we need a random floating point number from 0.0 up to 5.0. Because the length of this interval is 5.0, we multiply by this value (5.0 * *rand*) to stretch the interval of numbers. Mathematically we have the following:

$$0.0 \le rand < 1.0$$

Thus, multiplying by 5.0 throughout, we obtain the correct interval, as shown:

$$0.0 \le 5.0 * rand < 5.0$$

If the lower bound of the range is different from 0, we must also add that value. For example, if we need a random floating point number from 2.0 up to 7.0, we multiply by the length of the interval, 7.0 – 2.0 = 5.0, to expand the range. We then add the lower bound, 2.0, to shift or translate the result.

$$2.0 \le (7.0 - 2.0) * rand + 2.0 < 7.0$$

Generalizing, we can obtain a **uniformly distributed floating point number from *a* up to *b*** with the following computation:

(*b* - *a*) * *rand* + *a*

Quick Review Question 1 Start a new *M*-File. In opening comments, have "MATLAB Tutorial 4 Answers" and your name. Save the file under the name *MATLABTutorial4Ans.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. Generate a 3-by3 array of random floating point numbers between 0 and 1.
b. Generate a 1-by-5 array of random floating point numbers between 0 and 10.
c. Generate a 2-by-4 array of random floating point numbers between 8 and 12.

To obtain an integer random number, we can apply the ***floor*** function to the computation. This function returns the largest integer less than or equal to an argument. Thus, *floor*(2.8)and *floor*(2) are both 2. Because (7.0 – 2.0) * *rand* + 2.0 returns a random number from 2.0 up to but not including 7.0, the following expression returns an integer 2, 3, 4, 5, or 6, but not 7:

```
floor((7.0 – 2.0) * rand + 2.0)
```

Generalizing, we can obtain a **uniformly distributed integer from *a* up to but not including *b***, that is, integers in the set {a, a + 1, ..., b -1} with the following computation:

***floor*((*b* - *a*) * *rand* + *a*)**

Quick Review Question 2 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.

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. ***rand*('*state*', *n*)** seeds the random number generator with the integer n. For example, we seed the random number generator with 14234 as follows:

```
rand('state', 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. The following command seeds the random number generator with the time of day so that a different sequence of random numbers occurs for each run of a simulation:

```
rand('state', sum(100*clock))
```

clock returns a six-element array with numeric information about the date and time, including the seconds as a 2-digit number. We multiply the numbers in the array by 100 and find their sum to use as a seed for the random number generator.

Quick Review Question 3

a. Write a command to generate a list of ten 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 the last four digits of your Social Security Number or some other number as an argument. Execute the two commands in sequence several times, and notice that the list does not change.

Quick Review Question 4 Seed the random number generator with the time of day. Also, generate a table of 50 random integers between 4 and 20, inclusively, and assign the result to the variable y*1*. The result should be an array with values in the set {4, 5, ...20}.

Modulus Function

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

```
rem(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.

```
rem(23, 4)
```

Quick Review Question 5 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 *MATLAB* 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.

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

```
if condition
    tStatements
end
```

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
end
```

Alternatively, we can place the entire *if* statement on one line, ending each statement with a comma to display the result or semicolon to not display the result. One form of this command is as follows:

```
if condition tStatements, end
```

Thus, we can write the above *if* statement as follows:

```
if count ~= 0 total/count, end
```

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 *MATLAB* 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
end
```

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
end
```

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*.
For a one-line form of this *if-else* command, we use commas or semicolons at the end of each statement in the *if* body and *else* body, such as follows:

```
if x < y minxy = x, else minxy = y, end
```

Quick Review Question 6 Using the form that appears on several lines, 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.

Quick Review Question 7 Assign to a variable *cond* a random floating point number between 0 and 1. Revise your answer to Quick Review Question 6 to use a one-line *if* statement and to test if *cond* is less than 0.3. If you executed the statement a number of times, approximately what percentage of the time would you expect to obtain 1? Execute the statements 10 times, notice the relationship between the output values, and count the number of times 1 is the result of the *if* statement.

Still other forms of the *if* statement using an ***elseif*** clause to allow for more than two alternatives. The following two forms enable us to test for three choices:

```
if condition1
    body1Statements
elseif condition2
    body2Statements
else
    body3Statements
end
```

or

```
if condition1 body1Statements, elseif condition2 body2Statements, else body3Statements, end
```

With additional *elseif* classes, we can consider more alternatives.

Quick Review Question 8 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 arrays in *MATLAB*; 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 in the segment above that counts the number of random numbers less than 0.3. First, we generate a table of 0s and 1s, such that if a random number is less than 0.3, the table entry is 1. Then we count the elements in the table that match the pattern 1.

```
tbl = rand(1, 10) < 0.3
sum(tbl)
```

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

```
tbl =
     0   1   0   0   0   0   0   1   1   0
ans =
     3
```

Quick Review Question 9 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 10 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*** in this package returns the **mean**, or average, of the elements in an array and has the following format:

```
mean(list)
```

Similarly, ***std*** returns the **standard deviation** of the elements in a list. The following segment creates a list of 10 floating point numbers and returns the mean and standard deviation:

```
tbl = rand(1, 10)
mean(tbl)
std(tbl)
```

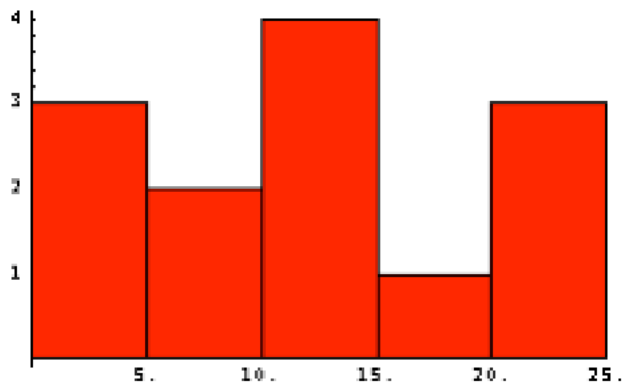
Quick Review Question 11

- **a.** Generate 10,000 **normally distributed random numbers** using the function ***randn*** with the same format as *rand*. Store the results in a vector, *normalNum*. Do not display *normalNum*.
- **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 $10 \leq x < 15$. 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 *MATLAB* 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 = [1, 15, 20, 1, 3, 11, 6, 5, 10, 13, 20, 14, 24];
hist(lst)
```

An optional second argument specifies the number of categories. Thus, the following command produces a histogram similar to that in Figure 1 with five intervals:

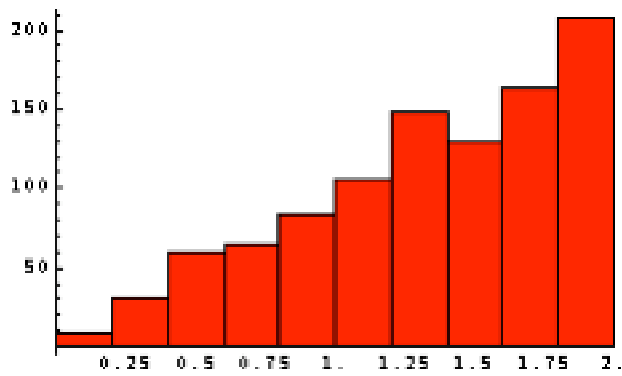
```
hist(lst, 5)
```

Figure 2 displays a histogram of an array, *tbl*, of 1000 random values from 0 to 2. (The table values only serve as data for graphing; each table entry is 2 times the maximum of two random numbers.) The commands to generate the table and produce the histogram are as follows:

```
tbl = 2 * max(rand(1, 1000), rand(1, 1000));
hist(tbl)
```

MATLAB uses 10 intervals, so each interval length is 0.25 (see Figure 2).

Figure 2 Histogram with 10 categories



Quick Review Question 12

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 5 categories.

d. Give the interval for the last category.

e. Approximate the number of values in this category.