

## 9.1 Computational Toolbox—Tools of the Trade: *Mathematica* Tutorial 4

File: *MathematicaTutorial4.nb*

*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

The prerequisites to this tutorial are *Mathematica* Tutorials 1-3. Tutorial 4 prepares you to use *Mathematica* for material in this and subsequent chapters. The tutorial introduces the following functions and concepts: random numbers, *Mod*, *If*, *Count*, *Flatten*, `<<` to load a package, *Remove*, *Mean* and *StandardDeviation* from the package *Statistics`DescriptiveStatistics`*, *Histogram* from the package *Graphics`Graphics`*, defining a package, truncating to an integer, and *ValueQ*.

Besides being a system with powerful commands, *Mathematica* 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 *Mathematica*. Besides more traditional applications, looping enables us to generate graphical animations that help in visualization of concepts. *Mathematica* is also easily extensible, and the tutorial considers how to load and use packages that extend the language. The statistical package is of particular interest in Module 9.3 on "Area through Monte Carlo Simulation." That module also employs several additional *Mathematica* commands, which this tutorial introduces.

- Do anything that is asked in Quick Review Questions, which appear in cells that look like this one. Because such cells are text cells and not input cells, do not type in these cells. Instead, move the cursor down until it changes from being vertical to being a horizontal-I bar. Click and start typing; a new input cell will form.

## 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.

*Mathematica* provides the random number generator **Random**. Each call to `Random[ ]` returns a uniformly distributed pseudorandom floating point number between 0 and 1. Execute the following cell several times to observe the generation of different random numbers:

```
Random[ ]
```

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

`Random` can have an argument of a type, such as **Real** or **Integer**, as with the following form:

```
Random[ type ]
```

With a default range of 0 to 1, execution of the following command with **Integer** as the only argument returns a random integer between 0 and 1:

```
Random[ Integer ]
```

Thus, we could use the command to simulate a random toss of a coin, with 0 indicating heads and 1, tails.

- **Quick Review Question 1**      **Generate a 3-by-3 matrix (list of 3 lists with 3 numbers each) of random 0s and 1s, and store the answer in *mat*. Change each occurrence of 0 to *RGBColor*[1, 1, 0] indicating yellow and each occurrence of 1 to *RGBColor*[0.1, 0.75, 0.2] for forest green, and store the result in *rgb*. Several modules, such as "Ant Movement," employ rule replacement of matrix values with *RGBColor* designations and then plot the result as a visualization of one time step of a simulation.**

An optional second argument specifies the range for the random numbers. If the second argument is a number (*max*), such as in the following form, the range is from 0 to that number:

```
Random[type, max]
```

The following command generates a random integer between 0 and 9; that is, the returned value is in the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

```
Random[Integer, 9]
```

The second argument can also be a list of two numbers, such as {*min*, *max*}, indicating the range (*min* to *max*), as in the following form:

```
Random[type, {min, max}]
```

The following command generates a random real (floating point) number between 2 and 9, such as 8.89723, 5.83752, and 2.8004:

```
Random[Real, {2, 9}]
```

- **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. **SeedRandom**[*n*] seeds the random number generator with the integer *n* in the square brackets. For example, we seed the random number generator with 14234 as follows:

```
SeedRandom[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 having no arguments for *SeedRandom* as follows, we seed the random number generator with the time of day and obtain a different sequence of random numbers for each run of a simulation:

```
SeedRandom []
```

### ■ Quick Review Question 3

- a. In a cell without *SeedRandom*, write a command to generate a list of ten random integers from 1 through 100, inclusively. Execute the cell several times, and notice that the list changes each time.
  
- b. Copy the cell from Part a. In the new cell before the command, call *SeedRandom* with the last four digits of your Social Security Number as an argument. Execute the cell 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 *y1*. The result should be a table with values in the set {4, 5, ...20}.

---

## Modulus Function

An algorithm for a random number generator often employs the modulus function, *Mod* in *Mathematica*, 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:

```
Mod[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.

```
Mod[23, 4]
```

- **Quick Review Question 5** Assign 10 to  $r$ . Then, assign to  $r$  the result of  $7r \bmod 11$ . 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 a segment 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 Module 2.1, the first *Mathematica* tutorial, we considered the function *Do*, which is one implementation of such a structure.

**Definition** 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.

*Mathematica* can implement the selection control structure with an **If** command. As with all *Mathematica* functions, square brackets, [ ], surround its arguments. One form of the function is as follows:

**If**[condition, t, f]

If *condition* has the value **True**, then the function returns the expression *t*; and if *condition* has the value **False**, *If* returns the expression *f*. Before executing the following commands, predict the output and the value of *minxy*:

```
x = 3;
y = 5;
minxy = If[x < y, x, y]
```

Because  $x$  is less than  $y$ , the *If* function returns the value of the second argument, 3. Thus, *minxy* is assigned that value. The *If* command accomplishes the same things as the following pseudocode:

if  $x$  is less than  $y$  then

```
    miny is assigned x  
else  
    miny is assigned y
```

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

If we do not want to save the value of the *If* function in a variable, such as *miny*, but only want to return the value, we do not use an assignment, such as follows:

```
x = 3;  
y = 5;  
If[x < y, x, y]
```

- **Quick Review Question 6** Write a command 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. If you executed the command a number of times, approximately what percentage of the time would you expect the function to return 1? Execute the command 10 times and count the number of times the function returns 1.
  
- **Quick Review Question 7** Copy the previous command and past it below the current cell. Revise the condition to store the value of the random number in a variable, *cond*. Press <RETURN> and type the name of the variable (*cond*). Thus, execution of the cell displays the result of the *If* function and the value of the random number, which *cond* stores. Execute the cell several times and notice the relationship between the output values.
  
- **Quick Review Question 8** Instead of executing the *If* function 10 times separately, we can automate the process using a table. Write a statement to return a table of 10 elements, where each element is the execution of the *If* command that returns 1 if a random floating point number is less than 0.3 and 0 otherwise. Thus, you can use the first *If* command you wrote for the expression in the *Table* function. Execute the cell several times and observe the changing results.
  
- **Quick Review Question 9** For this question, generate 10 random floating point numbers between 0 and 1 in a *Do* loop, and display how many of these numbers are less than 0.3. Begin by initializing a counting variable *counter* to be 0. Within the body of the *Do* loop, have an *If* statement that increments *counter* if a randomly generated number is less than 0.3. After the loop, type *counter* so that upon execution *Mathematica* displays the variable's final value, which is a count of random numbers less than 0.3.

---

## Counting

Frequently, we employ lists in *Mathematica*; and instead of using a *Do* loop, we can use the function **Count** to count items in the list that match a pattern. The format of the *Count* command is as follows:

**Count**[*list*, *pattern*]

The function returns a count of the elements in the list that match the pattern. As the segment below illustrates, *Count*

provides an alternative to a *Do* 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 = Table[If[Random[] < 0.3, 1, 0], {10}]
Count[tbl, 1]
```

- **Quick Review Question 10** Write a segment to generate a table of 20 random integers between 0 and 5 and with *Count* to return the number of table elements equal to 3.

## Flatten

A matrix, or rectangular array, of values consists of a list of lists. Sometimes, we wish to apply a function, such as *Count*, to all the elements of the list without consideration of sublists. That is, we want to eliminate internal braces. The function *Flatten*, whose format follows, returns a flattened list:

```
Flatten[list]
```

After generating a table of tables, the segment below returns the flattened list. However, because the second statement is not an assignment to *tbl2*, the value of *tbl2* remains the same as after execution of the first line.

```
tbl2 = Table[Random[Integer, {0, 3}], {4}, {4}]
Flatten[tbl2]
```

- **Quick Review Question 11** Write a command to count the number of 3's in *tbl2*.

## Loading a Package

*Mathematica* is an extensible system. We can define our own functions and use functions from packages written by others. A **package** is a *Mathematica* file consisting primarily of definitions that we can load into our system to create additional functionality. For example, the package *Statistics`DescriptiveStatistics`* contains definitions of descriptive statistical functions, such as *Mean* and *StandardDeviation*. When *Mathematica* opens, some packages are loaded automatically, but many others are not. We instruct *Mathematica* to **load a package** (*package*) using `<<`, as follows:

```
<< package
```

In the following command to load the descriptive statistics package, the package name contains the **back single quote** (`'`), which appears on the top left of the keyboard:

```
<< Statistics`DescriptiveStatistics`
```

The function `Mean` in this package returns the `mean`, or average, of the elements in a list and has the following format:

```
Mean[list]
```

Similarly, `StandardDeviation` 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 = Table[Random[], {10}]
Mean[tbl]
StandardDeviation[tbl]
```

If we attempt to reference a function from a package before loading the package, we must destroy the referenced version with `Remove` before *Mathematica* can recognize the package's version. The format of the `Remove` command to remove `name` is as follows:

```
Remove[name]
```

Thus, if we have used `Mean` before loading `Statistics`DescriptiveStatistics``, we must execute the following removal:

```
Remove[Mean]
```

## ■ Quick Review Question 12

- The function `AtomicWeight` to return the atomic weight of an element is in the package `Miscellaneous`ChemicalElements``. Without loading the package first, attempt to find the atomic weight of sodium (*Sodium*) by executing the function call below. Notice that output does not contain the desired results.

```
AtomicWeight[Sodium]
```

- b. Using the question mark, find information on *AtomicWeight*.
- c. Remove the Global versions of *AtomicWeight* and *Sodium*.
- d. Load the package *Miscellaneous`ChemicalElements`*.
- e. Find the atomic weight of sodium.

## 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, execution of the code below yields a histogram of the set  $lst = \{1, 15, 20, 1, 3, 11, 6, 5, 10, 13, 20, 14, 24\}$ . 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.

**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 *Mathematica* package *Graphics`Graphics`* contains a command, *Histogram*, to produce a histogram of a list of numbers. The following code loads the package, which we only must do once per session; assigns a value to *lst*; and displays its histogram:

```
<< Graphics`Graphics`
lst = {1, 15, 20, 1, 3, 11, 6, 5, 10, 13, 20, 14, 24};
Histogram[lst];
```

Execution of the code below displays a histogram of a table, *tbl*, of 1000 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 = Table[2 * Max[Random[], Random[]], {1000}];
Histogram[tbl];
```

*Mathematica* determines an appropriate number of categories, here 40 intervals, each of length 0.05. We can specify the number of categories with the *Histogram* option `HistogramCategories`. The following command indicates 10 categories for the same data, so each interval length is 0.25.

```
Histogram[tbl, HistogramCategories -> 10];
```

### ■ Quick Review Question 13

- a. Generate a table, *sinTbl*, of 1000 values of the sine of a random floating point number between 0 and  $\pi$ .
- b. Load the appropriate package for plotting a histogram.
- c. Display a histogram of *sinTbl*.
- d. Display a histogram of *sinTbl* with 5 categories.
- e. Give the interval for the last category.
- f. Approximate the number of values in this category.

---

## Defining a Package

*Mathematica* includes a number of packages with the software, and other packages are available from various sources. We, too, can develop our own package of function and constant definitions that we can load as needed. A package enables us to encapsulate related, thoroughly tested definitions; and several projects require the development of packages.

The name of a package has the extension `.m`, and we load a user-defined package as we do a system package with `<<`. However, we must be careful to give the full path name of our own packages so that *Mathematica* can locate the file. The following command loads the package *myPackage.m* from the current directory:

```
<< myPackage.m
```

Refer to this file for the form of a package.

Opening comments give the name, author, and date of the package. `BeginPackage` specifies the context of the package as `myPackage`` with a back quote appearing at the end of the package name.

The subsequent section contains documentation for the functions and constants in the package for use with `?`. Thus, after loading the package, we can type the following query about the function `tripleMean`:

```
? tripleMean
```

*Mathematica* returns the documentation. The package presents this documentation by having the function name, double colons, equals mark, and the description in quotation marks, as follows:

```
tripleMean::usage =
  "tripleMean[a, b, c] gives the mean of three arguments."
```

After this public section of the package comes the private one, which starts with the following `Begin` command with back quotes before and after `Private` as the argument:

```
Begin["`Private`"]
```

This section contains definitions of the functions and constants. `End[]` terminates the private section:

```
End[]
```

The entire package ends with the following command:

```
EndPackage[ ]
```

We do not want to save the file as a regular notebook but as a package. Therefore, under the *File* menu and `Save As Special...` submenu, we select `PackageFormat` and save the file with the extension `.m`.

**■ Quick Review Question 14**

- a. In the file *myPackage.m*, insert documentation for a function *myMin*, which returns the minimum of two arguments.
  
- b. Define *myMin* in that file.
  
- c. Load the file.
  
- d. Have *Mathematica* display the documentation for *myMin*.
  
- e. Test the function for several pairs of arguments.

---

## Truncating to an Integer

*Several exercises and projects from Module 9.2 on "Simulations" can use truncation.*

The module on "Simulations" discusses implementation of random number generators. One such generator that returns an integer must truncate, or chop off, the decimal fraction of a floating point number to return the integer part. As the following examples illustrate, the *Mathematica* function `IntegerPart` performs such truncation towards zero.

```
IntegerPart [ 3 . 8 ]
```

```
IntegerPart [ - 3 . 8 ]
```

- **Quick Review Question 15** Define a function *absFractionalPart* to return the fractional part of a number as a nonnegative floating point number. For example, *absFractionalPart*[3.469] and *absFractionalPart*[-3.469] should both return 0.469. Thus, if parameter *x* is nonnegative, *absFractionalPart* returns the difference in *x* and the truncation of *x* to an integer. If *x* is negative, the function returns the difference in the truncation of *x* and *x*. Test the function thoroughly.

## Existence of a Value

Several projects from Module 9.2 on "Simulations" can use the material from this section.

In defining our own functions, we may need to verify that a variable has a value. Frequently, we do not want use a variable unless it has a value. Perhaps, if it does not, we may want to give the variable a default value. Thus, we need some way to check for the existence of a value for a simple variable or function name. The *Mathematica* function *ValueQ* returns *True* if its argument has a value and *False* otherwise. Thus, the following commands test if *abc* and *xyz* have values.

```
abc := 3;  
ValueQ[abc]
```

```
ValueQ[xyz]
```

- **Quick Review Question 16**
  - a. Clear the value of *x*. Write an *if* statement that returns 0 if *x* does not have a value and returns 2 times the parameter otherwise.
  - b. Assign a value to *x*. Copy the *if* statement from Part a and execute the segment