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

File: *MapleTutorial4.mw*

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

Introduction

The prerequisites to this tutorial are *Maple* Tutorials 1-3. Tutorial 4 prepares you to use *Maple* for material in this and subsequent chapters. The tutorial introduces the following functions and concepts: random numbers, modulus, *if*-statement and operand, counting the number of occurrences of a pattern, flattening, loading a package, mean, standard deviation, histograms, defining packages and procedures, truncation, and variable number of parameters.

Besides being a system with powerful commands, *Maple* is a programming language. The first tutorial covered programming with the loop construct *for*. In this tutorial, we consider a command for another programming feature, the selection construct. *Maple* is easily extensible, and this tutorial also discusses how to load and use packages that extend the language.

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.

Maple provides the random number generator ***rand***. Each call to *rand*( ) returns a uniformly distributed pseudorandom nonnegative integer. Execute the following cell several times to observe the generation of different random numbers:

```
> rand();
```

An optional argument specifies the range for the random numbers. If the argument is a positive integer, *bound*, such as in the following form, the range is from 0 to *bound* - 1, and *rand* returns a procedure (function) to generate the random numbers:

```
rand(bound)
```

We assign the call of *rand* to a variable, such as *randomRange*, and then invoke the procedure with this name, such as *randomRange*().

For example, suppose we need a procedure to generate a random integer between 0 and 9; that is, the returned value of the procedure is in the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. The value might represent a sensor value for a simulation. To obtain such a procedure, we call *rand* with argument 10, which is one more than the maximum value, 9. We assign the call to a variable, *sensorProc*, as below. With a semicolon terminating the statement, we observe that *rand*(10) returns a procedure to generate a random value in 0..9, not a value itself.

```
> sensorProc := rand (10);
```

To obtain a random number in {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, we call the procedure using the name *sensorProc* along with empty parentheses because *sensorProc* is a function. Execute the following command several times and observe the returned values:

```
> sensorProc ();
```

To create a list of 20 such random integers, we place in brackets a call to the sequence command with argument *sensorProc*(), as follows:

```
> [seq(sensorProc() , i = 1..20)];
```

Quick Review Question 1 Do anything that is asked of you in cells that look like this one, marked as a Quick Review Question in boldface. Because such cells are text cells and not input cells, do not type in these cells. Instead, if a greater than prompt (>) does not appear in an otherwise empty execution group below, from the *Insert* menu, *Execution Group* submenu, select *After Cursor*. Alternatively, use the shortcut indicated on that menu or click the icon ([>) to "Insert executable *Maple* input after the current paragraph."

a. Assign to variable *headsOrTails* a procedure that generates a random 0 or 1, indicating heads or tails, respectively.

b. Using *headsOrTails* and *seq*, 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*.

c. Using *eval*, change each occurrence of 0 in *headsOrTails* to *yellow* and each occurrence of 1 to the RGB color with values 0.1, 0.75, and 0.2 for forest green, and store the result in variable *rgb*. Several modules, such as "Ant Movement," employ rule replacement of matrix values with RGB color designations and then plot the result as a visualization of one time step of a simulation.

The argument to *rand* can also be a range of integers *min..max*, such as -5..5, indicating the range (*min* through *max*), as in the following form:

```
rand(min..max)
```

The ***uniform*** function from the ***stats*** package ***random*** package, can return uniformly distributed random floating point numbers. We load the package as follows:

```
> with(stats[random]):
```

With no argument, ***uniform()*** returns a uniformly distributed random floating point number between 0.0 and 1.0, while ***uniform(n)*** returns *n* such numbers. Execute the following commands several times and observe the output:

```
> uniform();
```

```
> uniform(10);
```

As with other functions in packages, we can employ the long form of the name, ***stats*[*random, uniform*]**, instead of loading the package, as follows:

```
> stats[random, uniform]() ;
```

To generate a random floating point number in a more general range from *min* to *max*, we employ ***uniform*[*min..max*]**. The following long form of the function call generate 5 uniformly distributed random numbers between 22.3 and 39.4:

```
> stats[random, uniform[22.3, 39.4] ](5);
```

If *stats*[*random*] is loaded, the following short form of the command accomplishes the same task:

```
> uniform[22.3, 39.4] (5);
```

Quick Review Question 2

a. Give a segment to generate a number representing a random throw of a die with a return value of 1, 2, 3, 4, 5, or 6.

b. Using the long form, give a command to generate a random floating point voltage from 0.0 to 5.0.

c. Load the package and subpackage containing *uniform*.

d. Using the short form with no range specification, give a command to generate 3 uniformly distributed floating point numbers between 0.0 and 1.0.

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. The function call ***randomize(n)*** seeds the random number generator with the integer *n*. For example, we seed the random number generator with 14234 as follows:

```
> randomize(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 argument for *randomize*, as follows, we seed the random number generator with a number based on the system clock and obtain a different sequence of random numbers for each run of a simulation:

```
> randomize();
```

Quick Review Question 3

a. In a execution group without *randomize*, write a command to generate a list of ten random integers from 1 through 100, inclusively. Execute the execution group several times, and notice that the list changes each time.

b. Copy the execution group from Part a. In the new execution group before the command, call *randomize* with the last four digits of your Social Security Number as an argument. Execute the execution group 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 list with values in the set {4, 5, ...20}.**

Modulus

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

```
m mod 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 mod 4;
```

An alternative form employs a **mod function**, which surrounds *mod* with **backquotes (`)**, or grave accent characters, and has the *m* and *n* as arguments. The backward apostrophe key is towards the top left of the keyboard on the same key with tilde (~).

```
`mod`(m, n)
```

We employ the *mod* function instead of the *mod* operator when using a modulus result in an expression, such as an assignment statement. The following command using the *mod* function assigns the result of 23 *mod* 4 to *md*:

```
> md := `mod`(23, 4);
```

Quick Review Question 5 **Assign 10 to *r*. Then, assign to *r* the result of 7 *r* modulus 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 *Maple* tutorial, we considered the function *for*, which is one implementation of such a structure.

Definition **flow of control** 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.

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

```
if (condition) then
 trueStatementSeq
end if
```

If *condition* evaluates to be true, then *Maple* executes the statement(s) between *then* and *end if*. Otherwise, *Maple* skips this sequence and continues executing after the *if* statement. For example, perhaps we want to print every tenth value in a loop that has index *i*. After assignments to *i* and *datum* in the following abbreviated segment, we test if the remainder of *i* divided by 10 is equal to 0. If so, we display *i* and *datum*. Whether the remainder is 0 or not, the segment executes the *print* statement after the *if*. Execute this segment. Then, change the value of *i* to 3 and observe that the *print* within the *if* statement does not execute.

```
> i := 20:
  datum := 17:

  if (`mod`(i, 10) = 0 ) then
     print("In if, i = ", i, "datum = ", datum);
  end if;

  print("After if-statement");
```

The **equal sign (=)** that we used to test equality in the *if* condition above is an example of a relational operator. A **relational operator** is a symbol that we use to test the relationship between two expressions, such as two variables. *Maple* six relational operators, defined in the following table:

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

Operators with two characters must not contain spaces. The expression (*n* <> 7) means, “Is the value of the variable *n* not equal to 7?” The answer to this question obviously is either yes or no. In programming logic, however, we use the terms ***true*** and ***false*** instead.

Frequently, we have an "either-or" situation, which we program with another form of the *if*-statement, as follows:

```
if (condition) then
 trueStatementSeq
else
```

falseStatementSeq
end if

If *condition* has the value *true*, then *Maple* executes the statement sequence *trueStatementSeq*; and if *condition* has the value *false*, *Maple* executes the statement sequence *falseStatementSeq*. Before executing the following commands, predict the output and the value of *minxy*:

```
> x := 3:
  y := 5:
  if (x < y) then
     minxy := x
  else
     minxy := y
  end if;
```

Because *x* is less than *y*, the *if* construct executes the *then* statement assigning *x* to *minxy*. The *if* statement accomplishes the same things as the following pseudocode:

x isifless than *y* then
minxy is assigned *x*
else
minxy is assigned *y*

Quick Review Question 6 Write a segment to generate and test a uniformly distributed 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 segment 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.

Sometimes we need more than two choices. In this case, we use an ***else-if (elif)*** clause, as follows:

if (*condition1***) then**
statementSeq1
elif (*condition2***) then**
statementSeq2
else
statementSeq3

If *condition1* is *true*, *Maple* executes *statementSeq1*. If *condition1* is *false* but *condition2* is *true*, *Maple* executes *statementSeq2*. If both conditions are *false*, *Maple* executes *statementSeq3*.

Quick Review Question 7 Copy the previous answer and paste it below. Before the *if*-statement, store the value of the random number in a variable, *cond*, and end the assignment in a semicolon so that *Maple* displays the number. Revise the *if* statement so that if the random number *cond* stores is less than 0.3, print "less than 30%"; otherwise, if *cond* is less than 0.7, print "between 30% and 70%"; otherwise, print "70% or more". Execute the execution group several times, and notice the relationship between *cond* and the print-out.

In several of the simulations, such as using *if* in an assignment or a sequence statement, we must use an ***if operator*** instead of an *if* statement. Similar to the *mod* function, we place a backquote (`) before and after *if* and have arguments of a condition, *then* clause, and *else* clause. In the following general form, if the first argument, *condition*, is true, the operator form of *if* returns the second argument, *trueExpression*; but if *condition* is false, the expression returns the third argument, *falseExpression*.

`if`(*condition*, *trueExpression*, *falseExpression*)

Quick Review Question 8

a. Convert your answer to Quick Review Question 6 to an *if* operator form.

b. Copy your answer to Part a below and place it in a loop that executes 10 times, so that the loop displays 10 random 0's and 1's.

c. Instead of executing the *if* statement 10 times separately or using a loop, we can automate the process using a table (list). Write a statement to assign to *less30* a table of 10 elements, where each element is the execution of the *if* operator from Part a . Execute your answer 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 *for* 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 *for* 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 *Maple* displays the variable's final value, which is a count of random numbers less than 0.3.

Counting

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

numboccur[*list*, *pattern*]

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

```
> rand01 := proc()
     RandomTools[Generate](float(method = uniform))
  end proc:

  tbl := [seq(`if`(rand01() < 0.3, 1, 0) , i = 1..10)];
  numboccur(tbl, 1);
```

Quick Review Question 10 Write a segment to generate a table of 20 random integers between 0 and 5 and with *numboccur* 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 `+`, to all the elements of the list without consideration of sublists. That is, we want to eliminate internal brackets or flatten the list of lists into one list. In the ***ListTools*** package, the function ***Flatten***, whose format follows, returns a flattened list:

Flatten(*list*)

The following segment defines a uniform random number generator, *rand0to3*, for values in the set {0, 1,

2, 3}. The second statement assigns to *tbl2* a list of four lists of random values.

```
> rand0to3 := rand(0..3):
  tbl2 := [seq(
             [seq(rand0to3(), i = 1..4)],
          j = 1..4)];
```

The segment below returns a flattened list. However, because the second statement is not an assignment to *tbl2*, the value of *tbl2* remains the same as before execution of the segment.

```
> with(ListTools):
  Flatten(tbl2);
```

Quick Review Question 11 Write a command to return the sum of the numbers at any level in *tbl2*.

Loading a Package

Maple is an extensible system. We can define our own functions and use functions from packages written by others. A package is a *Maple* file consisting primarily of definitions that we can load into our system to create additional functionality. For example, to use the *Flatten* function, we loaded the *ListTools* package, which defines *Flatten*. Packages can be organized into **subpackages**. For example, among others, the **statistical package *stats*** contains subpackage ***describe*** for data analysis functions, such as ***mean*** and ***standarddeviation***; subpackage ***fit*** for linear regression functions, such as ***leastsquare***; subpackage ***random*** for random numbers in various probability distributions, which we cover in Module 9.3; and subpackage ***statplots*** for plotting functions, such as ***histogram***, which we cover shortly. We instruct *Maple* to load a package (*package*) using ***with***, as follows:

with(*package*)

After loading a package, we can load a subpackage using *with*. Alternatively, we can load the package and subpackage in one step, as follows:

with(*package*[*subpackage*])

The following input execution groups illustrate two ways to load the subpackage *describe* of the package *stats:*

```
> with(stats[describe]):
```

```
> with(stats):
  with(describe):
```

With a package and subpackage loaded, we can employ the following **short format** to call a function from the subpackage:

***function*(*arguments*)**

For example, the function ***mean*** in the ***describe*** subpackage returns the mean, or average, of the elements in a list and has the following short format:

mean(*list*)

Similarly, the function ***standarddeviation*** returns the standard deviation of the elements in a list. With the *stats* subpackage *describe* loaded, the following segment creates a list of 10 integers between 0 and 99 and returns the mean and standard deviation:

```
> rand0to99 := rand(0..99):
  tbl := [seq(rand0to99(), i = 1..10)];
  mean(tbl);
  standarddeviation(tbl);
```

For a function in a subpackage, we can also use the following **long form** of a function call:

***package*[*subpackage*, *function*](*arguments*)**

For example, we can call *mean* as follows:

```
> stats[describe, mean](tbl);
```

Should the package be loaded, the following long form of a subpackage function can be written:

***subpackage*[*function*](*arguments*)**

In this situation, the call to *mean* is as follows:

```
> describe[mean](tbl);
```

If we attempt to reference a function from a package before loading the package, *Maple* substitutes what it can and returns the function call, such as in the following:

```
> histogram(tbl);
```

Realizing our omission, we can load the proper package and perhaps subpackage and re-execute the command, or we can employ the long form of the function call.

Quick Review Question 12

a. The function *GetElement* to return the atomic weight and other information about an element is in the package *ScientificConstants*. Without loading the package first, attempt to find information on sodium (*Sodium*) by executing the function call to *GetElement* below. Notice that output does not contain the desired results.

```
> GetElement(Sodium);
```

b. Using the question mark, find information on *GetElement*.

c. Using the long form of a call to *GetElement*, find information about sodium.

d. From the output, determine the atomic weight of sodium.

e. Load the package *ScientificConstants*.

f. Repeat Part c using the short form of the function call.

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 data in *lst* = [1, 15, 20, 1, 3, 11, 6, 5, 10, 13, 20, 14, 24]. In the figure, the 13 data values are split into three bars that have the same area. Because five data values (1, 1, 3, 6, and 5)

appear in the first interval, the height of that bar is 5.

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 *Maple* subpackage ***statplots*** in package ***stats*** contains a command, ***histogram***, to produce a histogram of a list of numbers. The code below loads the package, which we only must do once per session; assigns a value to *lst*; and displays its histogram with approximately equal areas for the bars. With the option-value pair ***area = count***, the height of each bar represents the number of data items in that category. Thus, five data items--1, 1, 3, 6, and 5--are between 1 and 8, inclusively.

```
> with(stats[statplots]):
```

```
> lst := [1, 15, 20, 1, 3, 11, 6, 5, 10, 13, 20, 14, 24]:
  histogram(lst, area = count);
```

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 floating point numbers between 0.0 and 1.0.) The commands to generate the table and produce the histogram are as follows:

```
> with(RandomTools):
  tbl := [seq(
      2 * max(Generate(float(method = uniform)) ,
    Generate(float(method = uniform)) ),
      i = 1..1000)]:
  histogram(tbl, area = count);
```

Maple determines an appropriate number of categories, here 12 intervals, each of length about 0.0833 units. We can specify the number of categories with the option ***numbars***. The following command indicates 10 categories for the same data:

```
> histogram(tbl, area = count,  numbars = 10 );
```

Quick Review Question 13

a. Generate a table, *Tbl*, 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 *Tbl*.

d. Display a histogram of *Tbl* with 5 categories.

e. Give the interval for the last category.

f. Approximate the number of values in this category.

Defining a Package

Maple 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.

We store packages in a **repository**, or **library**. We use the ***Maple library archive manager*** command ***maple*** to create and manipulate a repository. For creation, we have a leading argument of ***'create'***. The second argument is the path name for the archive in quotation marks. The abbreviation for the **current**

working directory is a period (.), and the extension for the repository is **.lib**. The third argument is an integer, such as 100, giving the approximate number of archive members, such as functions and constants. The command below creates in the current working directory library archive files *myLib.lib* and *myLib.ind* that can store about 100 members. If a repository already exists in this directory, *march* does not create a new one.

```
> march('create', "./myLib.lib", 100);
```

A package definition begins in a *Maple* worksheet with the name of the package, such as *myPackage* below, on the left-hand-side of an assignment statement. The right-hand-side of the definition below begins with ***module()*** and ends with ***end module***. An ***export*** statement lists the package functions and constants that are public so that code outside the package can access them. Package functions and constants not listed in the *export* statement cannot be used outside the *module* body. Opening comments give the name, author, and date of the package. Afterwards, we have the definitions of two functions, *tripleMean* and *myMax*. The constant *ZZZ* is private. Typically, a private function or constant is for local use by other functions. However, here, we use *ZZZ* only for illustration.

```
> myPackage := module()
  export tripleMean, myMax;

  #:Name: myPackage
  #:Author: me
  #:Date: now

  # tripleMean(a, b, c) gives mean of 3 arguments
  tripleMean := (a, b, c)->(a + b + c)/3.0;

  # myMax returns the maximum of two arguments
  myMax := (a, b)->`if`(a > b, a, b);

  # ZZZ is a constant
  ZZZ := 3;

  end module;
```

After defining this module, we need to save *myPackage*. The variable ***libname*** stores a sequence of strings for the file-system locations of the main *Maple* library and other libraries. To save *myPackage* in a directory not already in the sequence, we make a string with the location's name the first element of *libname* by assigning to *libname* the sequence of this string and the previous value of *libname*. Then, we employ the ***savelib*** command with the name of the module in apostrophes. In the example below, we adjust *libname* to first contain a string representing the current working directory, ".", and then we save *myPackage* to the repository, *myLib*, in that directory. *Maple* saves the package as *myPackage.m* in the repository *myLib*.

```
> libname := ".", libname;
  savelib('myPackage'):
```

To view the contents of an archive, we invoke *Maple* archive manager *march* with a ***'list' option*** and the repository. The manager returns a list with a list for each *.m* file in the repository along with its date and time, among other things. Thus, the output of the following command indicates that *myPackage.m* is a member of *myLib:*

```
> march('list', "./myLib.lib");
```

To observe how to load and use a package, we first restart the system.

```
> restart:
```

With the restart, the previous expanded value of *libname* has been lost. Thus, so that *Maple* can find our repository, we again place the string name of the current directory as the first member of the sequence of locations that *libname* holds. We load a user-defined package as we do a system package, using *with*. The command returns a list with the functions that are publicly available for use.

```
> libname := ".", libname:
  with(myPackage);
```

The following two statements illustrate that we can use *tripleMean* but not the value of *ZZZ:*

```
> tripleMean(1, 3, 6);
```

```
> ZZZ;
```

Quick Review Question 14

a. Restart the system.

v. If you did not do so above, in your current directory, create a repository *myLib*.

c. Define a package *myPkg* that contains a public function, *myMin*, which returns the minimum of two arguments.

d. Save the package to your repository.

e. List the contents of your repository.

f. Restart the system.

g. Load the package *myPkg*.

h. Test the function for several pairs of arguments.

Procedure Definitions

In Tutorial 1, we discussed defining simple functions that have one statement. Frequently, however, the function needs a group of several statements. To do so, we employ a **procedure**. As with the format we have been using, the function name is on the left-hand-side of an assignment equal. The right-hand-side begins with ***proc*** and ends with ***end proc***. After *proc*, parentheses surround parameters. In longer function definitions, we often employ temporary variables that should only be active within the definition. To avoid conflicts with variable names elsewhere, we make the variables local to the definition with a ***local*** and list of local variables. Another optional statement is ***description***, which contains a comment in quotation marks. When a procedure is displayed, so is its description.

The procedure below returns a list with the floating point quadratic formula values. A quadratic formula, $(-b \pm \sqrt{b^2 - 4\,a\,c})/(2a)$, provides the solution(s) to a quadratic equation, $a\,x^2 + b\,x + c = 0$. If the determinant, $b^2 - 4\,a\,c$, is zero, only one solution, $-b/(2a)$, exists in the set of real numbers. If the determinant is positive, two distinct solutions exist. However, if the determinant is negative, no real number solution exists. The procedure definition of *quadraticFormula* below has three parameters, *a*, *b*, and *c*, that are the coefficients in the quadratic equation. A local variable, *discriminant*, is unknown outside the function definition. The following definition illustrates the form of a function definition using *proc*, *local*, and *description:*

```
> quadraticFormula := proc(a, b, c)
     local discriminant;
     description "Quadratic Formula";
     discriminant := b^2 - 4 * a * c;
     if (discriminant > 0) then
        [(-b + sqrt(term))/(2 * a),
         (-b - sqrt(term))/(2 * a)];
     elif (discriminant = 0) then
        [-b/(2 * a)];
```

```
    else
       [];
    end if;
  end proc:
```

The following call to *quadraticFormula* returns a list with the two distinct solutions to $x^2 + x - 6 = 0$:

```
> quadraticFormula(1, 1, -6);
```

Because the discriminant is zero, the following call returns a list with the one distinct solution to $x^2 - 6x + 9 = 0$:

```
> quadraticFormula(1, -6, 9);
```

With a negative discriminant, the quadratic equation has no real solutions, so the following call returns an empty list $3x^2 + 5x + 7 = 0$:

```
> quadraticFormula(3, 5, 7);
```

Although the function definition assigns a value to the local variable *discriminant*, we cannot access this value outside the definition, as the following illustrates:

```
> discriminant;
```

Occasionally, such as in several simulations for Chapter 11, to avoid passing a widely used value through the argument-parameter list, it is convenient to employ a variable that is global. In such a case, we declare the variable to be ***global***. However, such circumstances are rare. Thus, we should use the feature sparingly and should carefully document use of any global variable. The following command declares variables *rate* and *probabilityOfFall* to be global:

```
> global rate, probabilityOfFall;
```

Quick Review Question 15

a. Define the function *myMax* from the section on "Defining Packages" to be a procedure. Have a procedure description. Employ a local variable *returnMax*. Use an *if* statement instead of an *if* operator. Inside the body of the *if* statement, assign the return value to *returnMax*. Have a statement with *returnMax;* on a line by itself after the *if* statement.

b. Test the function thoroughly.

Truncation

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 *Maple* function ***trunc*** performs such truncation towards zero.

```
> trunc(3.8);
```

```
> trunc(-3.4);
```

Quick Review Question 16

a. 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*. Define *absFractionalPart* using *proc*.

b. Test the function thoroughly.

Variable Number of Parameters

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

In defining our own functions, we may need a variable number of arguments for a procedure. Perhaps, if an argument is missing, we may want to give the variable a default value. Thus, we need some way to represent and count the number of arguments. To define such a function, we do not indicate any parameters. Within a procedure, ***nargs*** gives the number of arguments; and we use ***args*[1]** for the first argument, ***args*[2]** for the second, and so forth. The following procedure returns the maximum of exactly two arguments. For exactly one argument, the procedure returns that argument. In all other cases, the procedure returns 0.

```
> max2 := proc()
    local returnVal;
    description " Returns max of 2 arguments, one argument, or 0";
    if (nargs = 1) then
       returnVal := args[1]
    elif (nargs = 2) then
       if (args[1] > args[2]) then
          returnVal := args[1]
       else
          returnVal := args[2]
       end if
    else
       returnVal := 0
    end if;
    returnVal;
 end proc:
```

```
> max2();
```

```
> max2(5);
```

```
> max2(7, 9);
```

```
> max2(17, 9);
```

```
> max2(17, 9, 4);
```

Quick Review Question 17

a. Define a function *sumAll* that returns 0 if the function has no arguments and otherwise returns the sum of all arguments. Use *proc*. For the sum of several arguments, initialize a local variable, say *returnVal*, to be 0. Then, within a *for* loop having index *i*, assign to *returnVal* the sum of *returnVal* and *args*[*i*].

b. Thoroughly test the function.