10.1 Computational Toolbox—Tools of the Trade: *Maple* Tutorial 5

File: ***MapleTutorial5.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 1-4. Tutorial 5 prepares you to use *Maple* for material in this and subsequent chapters. The tutorial introduces the following functions and concepts: taking part of a list, maximum and minimum, delayed plotting, animation, and 3D point graphics.

Taking Part of a List

Instead of obtaining a longer list with new values appended onto the end, sometimes we need to get a smaller list consisting of the first few elements in the original list. The following form returns the sublist of *List* consisting of the first *n* elements:

```
List[1..n]
```

Thus, in the following segment, without changing the value of a list (*lst*), we obtain a list of the first 3 character elements.

```
> lst:=["a","b","c","d","e","f"]:
  lst[1..3];
  lst;
```

The following form with a range of negative integers returns a sublist of *List* consisting of the last *n* elements:

```
List[-n..-1]
```

Thus, the command to return the sublist of the last 3 character elements of *lst* follows:

```
> lst[-3..-1];
```

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."

The assignment statement in following segment generates three random points with coordinates between 0 and 1. The last statement plots the points as fairly large blue points in a 1-by-1 rectangle. Adjust the code to generate three plots. The first graphics only displays the first point; the second graphics shows the first two points; and the third, all three points. Later in this tutorial, we discuss how to animate the plots so as to show the points appearing one at a time. For

such an animation, why would we want to specify the plot viewing area to be the same for each graphic?

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

```
> pts := [seq([uniform(), uniform()], i = 1..3)]:
```

```
> plots[listplot](pts, style = point,
      symbolsize = 20, color = blue,
      view = [0..1, 0..1]);
```

Maximum and Minimum

The *Maple* function ***max*** returns the maximum of numeric arguments. Thus, a call to the function has the following form:

max(*x1*, *x2*, ..., *xn*)

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

```
> max(2, -4, 7, 3);
```

In several applications, we need to obtain the maximum value in a list of numbers. To do so, we obtain the sequence of numbers using *op* and take *max* of the result. In the following segment, *max* returns the maximum value in a list:

```
> lst := [-1, 0, -1, 0, 1, 2, 3, 2, 1, 0]:
  max(op(lst));
```

When we need to find the maximum number in several lists, we perform *op* on each list, as in the following example:

```
> lst2 := [3, 7, -5]:
```

```
> max(op(lst), op(lst2) );
```

Similar to the *max*, the *Maple* function ***min*** returns the minimum of its numeric arguments.

Quick Review Question 2 Write a segment to generate but not display a list of 100 random floating point numbers between 0 and 1 and to display the maximum and minimum values in the list.

Delayed Plotting

We often wish to show two plots on the same graph. In Module 8.3 on "Empirical Models," we display a set of data points along with a curve that captures the trend of the data. For example, the segment below plots a set of points, a regression line, and the two graphics together. By assigning the first two graphs to variables and using colons to terminate the commands, we are able to suppress intermediate plots and output. Then, the *display* command causes the two plots to appear in one display.

```
> pts := [[0.2, 0.1], [0.4, 0.3], [0.3, 0.3], [0.3, 0.6]]:
  with(plots):
  lp := listplot(pts, style = point,
        symbolsize = 20, labels = [" x "," y "],
        view = [0..0.62, 0..0.62]) :
  pltfit := plot(0.025 + x, x = 0..0.62) :
  display(lp, pltfit);
```

Quick Review Question 3 The first statement of following segment generates 10 random points

with coordinates between 0 and 1. The two *listplot* commands are identical. Sometimes, we wish to display points prominently as well as to have line segments connecting the points. Adjust the second plot so that black (not blue) line segments connect adjacent points in the sequence. Also, adjust each graphics command and add another command so that *Maple* only displays a final combined plot of blue points and black line segments.

```
> pts := [seq([uniform(), uniform()], i = 1..10)]:
  with(plots):
  listplot(pts, style = point,
        symbolsize = 20, color = blue,
        view = [0..1, 0..1]);
  listplot(pts, style = point,
        symbolsize = 20, color = blue,
        view = [0..1, 0..1]);
```

Animation

Animations are useful and interesting for visualizing computer simulation results. To create an animation, we generate a list containing a sequence of plots, as below. In making the graphics, plots should have the same *view* value so that axes appear fixed during the animation.

```
> sineAnim := [seq( plot(i * sin(t), t = 0..2 * Pi),
      i = 1..10):)
```

To prepare the animation, we use *display* from the *plots* package along with the variable (*sineAnim*) storing the plot list as an argument along with the option-value pair ***insequence = true***. The text below explains how to animate the graphics that results from executing the following command:

```
> plots[display](sineAnim, insequence = true);
```

To animate, we first click on the above output so that a box surrounds the graphics and icons appear on the top left of the window. We can click on these icons to regulate the animation during execution, as follows:

- square - stop animation
- right-pointing triangle - play animation
- right-pointing triangle and rectangle - show next frame

rectangle and left-pointing triangle - display graphics in reverse order of appearance

- rectangle and right-pointing triangle - display graphics in order of appearance
- loop with arrow head on top - run one cycle
- loop - cycle through graphics repeatedly
- double left-pointing triangles - slow down animation
- double right-pointing triangles - speed up animation
- FPS - records frames per second

After clicking a directional icon, such as *loop*, we still must click the *play* icon to view the animation.

Quick Review Question 4 The function *f* below is the logistic function for constrained growth, where the initial population is 20, the carrying capacity is 1000, and the rate of change of the population is (1 + 0.2*i*) (see Module 3.4 on "Constrained Growth"). To see the effect of increasing the rate of change from (1 + 0.2(1)) = 1.2 to (1 + 0.2(10)) = 3.0, complete the command to generate a list of 10 plots of *f*(*t*, *i*), where *i* varies from 1 to 10. Display the animation, and regulate the animation using each of the icons on the top left of the window.

```
> unassign(f):
  i = 'i':
  f := (t, i)->
   (1000*20)/((1000-20)*exp(-(1 + 0.2*i)*t) + 20);
```

```
> plot(f(t, i), t = 0..3, y = 0..1000);
```

3D Point Graphics

Project 9 from Module 10.2 on "Random Walk" and some projects in Chapter 11 use the material from this section.

For three-dimensional (3D) graphics images points, we employ the *plots* package function ***pointplot3d***. Instead of two coordinates, points in 3D have three coordinates. The following command displays a 3D graphics image of three points, which are in a list:

```
> plots[pointplot3d] ([[0,0,1], [2, 3, 4], [3, 1, 2]],
      style = point,
      symbol = circle,
       symbolsize = 20, color = blue);
```

We can rotate by clicking on the graphics and adjusting the icons on the top left of the window.

Quick Review Question 5 Copy the first execution group from Quick Review Question 1 to below. Adjust the assignment statement to assign to *pts* ten random 3D points with coordinates between 0 and 1. Assign to *lseqPlot* a list of ten 3D point plots, where the first plot shows the first point; the second, the first two points; and so forth. Animate the plots, and use the icons at the top left of the window to regulate the animation.

Logical Operators

The material in this section is useful for several projects in Chapter 13 that are appropriate after covering the current chapter.

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

```
> x := 6:
  if (x < -3) or (x > 3) then
      print("Out of bounds")
  end if;
```

The opposite condition, $-3 \le x \le 3$, requires the element-by-element AND operator ***and***, as in the following example:

```
> x := 1:
  if (-3 <= x) and (x <= 3) then
      print("In bounds")
  end if;
```

We cannot write the condition as in mathematics, $(-3 \le x \le 3)$, but must express the condition with a compound statement.

To negate a condition, we employ the element-by-element NOT operator ***not***, Thus, not($x > 3$) is equivalent to ($x <= 3$). ***not***, ***and***, and ***or*** are called **logical operators**.

Expressions can involve logical operators in conjunction with arithmetic and relational operators. In such cases, the operator precedence of Table 1 determines the order of evaluation. When in doubt, we can always use parentheses to clarify the precedence as in the above two if statements. However, as Table 1

indicates, we can omit the parentheses, as follows, because *Maple* evaluates the two inequalities before evaluating the OR operator:

```
> x := 6:
  if x < -3 or x > 3 then
      print("Out of bounds")
  end if;
```

Table 1 Operator precedence from highest to lowest

1. ()
2. ^
3. Unary -
4. * /
5. < <= > >= = <>
6. *not*
7. *and*
8. *or*
9. :=

Quick Review Question 6 **Write an *if* statement for the following situation and test using several values for the variables: If *x* + 2 is greater than 3 or *y* is less than *x*, add 1 to *y*; otherwise, subtract 1 from *x*.**

Membership

The material in this section is useful for several projects in Chapter 13 that are appropriate after covering the current chapter.

The boolean function ***member*** determines if an expression is a member of an array or not. A general form is as follows:

member(*expr*, *list*)

The command returns *true* if *expr* is a member of *list* and *false* otherwise.

Quick Review Question 7 **Write an *if* statement for the following situation and test using several values for the variables: If *x* is an element of list *v*, display "Is a member"; otherwise, display "Is not a member".**

While Loop

The material in this section is useful for several projects in Chapter 13 that are appropriate after covering the current chapter.

We have employed the *for* loop to repeat a segment of code when we know the number of iterations. However, if a loop must execute as long as a condition is true, we can use a ***while*** loop. The form of the command is as follows:

```
while condition do
 statement(s)
end do
```

For example, the segment below generates and displays random numbers between 0.0 and 1.0 as long as the values are less than 0.7. The segment also counts how many of the random values are in that range.

```
> counter := 0:
  ra := uniform();
  while ra < 0.7 do
     counter := counter + 1;
     ra := uniform();
     print("ra: ", ra);
  end do:
  counter;
```

We initialize to zero a variable, *counter*, that is to count the number of random numbers less than 0.7. Before the loop begins, we prime *ra* with a random number so that *ra* has an initial value to compare with 0.7. Then, at the end of the loop, we obtain and display another value for *ra* to compare with 0.7. After completion of the loop, we display the final value of counter.

Quick Review Question 8 Write a segment to generate an animation, as follows: Assign 0 to *x*. While *x* is between -5 and 5, plot the point (*x*, 0) as a large red dot; generate a random integer -1, 0, or 1; and assign to *x* the sum of this number and *x*.