

10.1 Computational Toolbox—Tools of the Trade: *Mathematica* Tutorial

5 (v. 7)

File: *MathematicaTutorial5.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
Tutorial © 2009

Introduction

The prerequisites to this tutorial are *Mathematica* Tutorials 1-4. Tutorial 5 prepares you to use *Mathematica* for material in this and subsequent chapters. The tutorial introduces the following functions and concepts: *Take*, *Max* and *Min*, *AxesOrigin*, animation, 3D point graphics, and several features for projects, including the *While* loop, logical operators, and testing membership.

- 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.

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 of a call to the function *Take* returns the sublist of *list* consisting of the first *n* elements:

```
Take[list, n]
```

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

```
lst = {"a", "b", "c", "d", "e", "f"};  
Take[lst, 3]  
lst
```

The following form of a call to *Take* with a negative integer as the second argument returns the sublist of list consisting of the last n elements:

```
Take[list, -n]
```

Thus, without changing the value of list (*lst*), *Take* returns the sublist of the last 3 character elements of *lst* in the following:

```
Take[lst, -3]
```

- **Quick Review Question 1** The first statement of following segment generates 3 random points with coordinates between 0 and 1. The second plots the points as fairly large blue dots in a 1-by-1 rectangle. Adjust the code to generate 3 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?

```
pts = Table[RandomReal[{0, 1}, 2], {3}];
ListPlot[pts, PlotStyle -> {PointSize[0.03], RGBColor[0, 0, 1]},
PlotRange -> {{0, 1}, {0, 1}}]
```

Maximum and Minimum

The *Mathematica* function *Max* returns the maximum of numeric arguments or of one or more lists of numeric arguments. Thus, one form of a call to the function can be as follows:

```
Max[x1, x2, ... ]
```

In this case, the function returns the numerically largest of x_1, x_2, \dots . For example, the following invocation with four arguments returns the maximum argument, 7.

```
Max[2, -4, 7, 3]
```

Another form of a call to the function, which follows, has lists of numbers as arguments:

```
Max[{x1, x2, ...}, {y1, y2, ...}, ...]
```

In the following segment, *Max* returns the maximum value in a list.

```
lst = {-1, 0, -1, 0, 1, 2, 3, 2, 1, 0};
Max[lst]
```

Similarly, the *Mathematica* function *Min* returns the minimum of numeric arguments or of one or more lists of 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 in the list.

Axes Intersection

The previous graphics example might be clearer if the axes intersected at the origin. For animations, too, the axes must cross in the same place on each plot. To specify the point $\{x, y\}$ at which the axes intersect, we use the plot option `AxesOrigin`, as `AxesOrigin -> {x,y}`. The segment below draws the graph above with the axes crossing at the origin. A gap on the t axis appears between 0 and 1 because t values begin at 1 in this example..

```
ylst = {-1, 0, -1, 0, 1, 2, 3, 2, 1, 0};
lp = ListPlot[ylst, AxesLabel -> {"t", "y"}, Joined -> True,
  AxesOrigin -> {0, 0}]
```

- **Quick Review Question 3** Adjust the above code to produce a line plot of 5 random points so that the axes cross at the origin.

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 following segment plots a set of points, a regression line, and the two graphics together.

```
pts = {{0.2, 0.1}, {0.4, 0.3}, {0.3, 0.3}, {0.3, 0.6}};
lp = ListPlot[pts, PlotStyle -> {PointSize[0.03]},
  AxesLabel -> {" x ", " y "}, PlotRange -> {{0, 0.62}, {0, 0.62}},
  AspectRatio -> 1]
pltfit = Plot[0.025 + x, {x, 0, 0.62}, AspectRatio -> 1]
Show[lp, pltfit]
```

Sometimes, we do not want to show the separate plots. To generate graphics but suppress the display, we include the `semicolon`. Then, in the `Show` command, we instruct *Mathematica* to generate the display without the semicolon. The following segment, which generates only one display, is as above with the exception of the semicolons after the `ListPlot` and `Plot` commands.

```
pts = {{0.2, 0.1}, {0.4, 0.3}, {0.3, 0.3}, {0.3, 0.6}};
lp = ListPlot[pts,
  PlotStyle → {RGBColor[1, 0, 0], PointSize[.03]},
  AxesLabel → {" x ", " y "},
  PlotRange → {{0, 0.62}, {0, 0.62}},
  AspectRatio → 1];
pltfit = Plot[0.025 + x, {x, 0, .62}, AspectRatio → 1];
Show[lp, pltfit]
```

- Quick Review Question 4** The first statement of following segment generates 10 random points with coordinates between 0 and 1. The *ListPlot* for *lp* and *lpLine* are identical, and *Mathematica* displays three graphs. 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, have *Mathematica* only display the final combined plot of blue points and black line segments.

```
pts = Table[RandomReal[{0, 1}, 2], {10}];
lp = ListPlot[pts,
  PlotStyle → {PointSize[0.03], RGBColor[0, 0, 1]},
  PlotRange → {{0, 1}, {0, 1}}]
lpLine = ListPlot[pts,
  PlotStyle → {PointSize[0.03], RGBColor[0, 0, 1]},
  PlotRange → {{0, 1}, {0, 1}}]
Show[lp, lpLine]
```

Animation

One interesting use of the *Do* loop is to draw several plots for an animation. In making the graphics, plots should have the same *PlotRange* so that axes appear fixed during the animation. After generating the plots, we can animate the sequence by calling *ListAnimate* with the list of plots to animate. To animate, we store the graphics to animate in a list, say *lst*, and then call *ListAnimate[lst]*. Clicking on pause stops the animation. To regulate the animation during execution, we can click on the following icons on the top of the cell:

- right arrow - displays the graphics in order of appearance
- parallel vertical lines - toggles between pausing and continuing
- double down arrow - slows down the animation
- double up arrow - speeds up the animation

We can also animate a function by calling *Animate* as follows:

Animate[*expr*, {*u*, *umin*, *umax*}]

u is the variable that changes from frame to frame.

- **Quick Review Question 5** 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 $(1 + 0.2i)$ (see Module 3.3 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 10 plots of $f[t, i]$, where *i* varies from 1 to 10, by enclosing the *Plot* command in an *Animate* function. Execute the command, and regulate the animation using each of the icons on the top of the cell.

```
Clear[f]

f[t_, i_] := 
$$\frac{1000 * 20}{(1000 - 20) * \text{Exp}[-(1 + 0.2 i) * t] + 20}$$


i = 10;
Plot[f[t, i], {t, 0, 3}, PlotRange -> {{0, 3}, {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.

In "Mathematica Tutorial 3" (Module 8.1), we discussed that *Graphics* returns a two-dimensional graphics image and that *Line* is one of the graphics primitives. For three-dimensional graphics images, we employ *Graphics3D*. Like *Line*, *Point* is a primitive in 2D or 3D graphics. However, instead of two coordinates, locations in 3D have three coordinates. For example, *Point*[{0, 0, 0}] is a point at the origin. (*Graphics3D*, *Point*, and other 3D commands, directives and primitives are available under *Classroom Assistant, 3D*.) The following command with square brackets enclosing braces generates a 3D graphics image containing a point at the origin whose diameter is 0.1 of the width of the graphics:

```
Graphics3D[{PointSize[0.1], Point[{0, 0, 0}]}]
```

We can follow the *Graphics3D* command above with a comma and the *PlotRange* option to indicate minimum and maximum *x*, *y*, and *z* values. The form of this option is as with 2D *Graphics* (see "Mathematica Tutorial 3") but can include a third pair for the minimum and maximum *z* values.

- **Quick Review Question 6** 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. Initialize *lst* to be an empty list. Store ten 3D point plots in *lst*, where the first plot shows the first point; the second, the first two points; and so forth. Animate the plots with *ListAnimate*, and use the icons at the top of the cell 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 **OR operator** in *Mathematica* is **||**, so the statement is as follows:

```
x = 7;
If[(x < -3) || (x > 3), "Out of bounds"]
```

The opposite condition, $-3 \leq x \leq 3$, requires the **AND operator** **&&**, as in the following example:

```
x = 2;
If[(-3 ≤ x) && (x ≤ 3), "In bounds"]
```

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

To negate a condition, we employ the **NOT operator** `!`. Thus, `!(x > 3)` is equivalent to `(x <= 3)`. `||`, `&&`, and `!` 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 *Mathematica* evaluates the two inequalities before evaluating the OR operator:

```
x = 7;
If[x < -3 || x > 3, "Out of bounds"]
```

Table 1 Operator precedence from highest to lowest

1. `()`
2. `^`
3. Unary `-`
4. `*` `/`
5. `<` `<=` `>` `>=` `=` `!=`
6. `!`
7. `&&`
8. `||`
9. `=`

- **Quick Review Question 7** 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 `MemberQ` determines if an expression is a member of a list or not. A general form is as follows:

```
MemberQ[list, expr]
```

The command returns *True* if *expr* is a member of *list* and *False* otherwise.

- **Quick Review Question 8** 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, body]
```

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 = RandomReal[]
While[ra < 0.7,
  counter = counter + 1;
  ra = RandomReal[];
  Print["ra: ", ra]
]
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 9** Write a segment to generate an animation, as follows: Assign 0 to *x* and an empty list to *lst*. While *x* is between -5 and 5, append to *lst* a *ListPlot* of 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*. After the loop, animate the list of plots.