11.1 Computational Toolbox—Tools of the Trade: *Maple* Tutorial 6

File: *MapleTutorial6.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 1-5. Tutorial 6 introduces the following features, which simulations of this chapter employ: joining sublists, length of a list, two methods for visualizing grids, types, and defining rules.

Joining Sublists

By specifying a range of indices, we can obtain a sublist of a list. For example, *lst*[1..3] returns a sublist with the first three elements of list *lst*. Moreover, *op*(*lst*[1..3]) returns a sequence of these elements. For example, the following command defines a list , *alphas*, of strings with letters.

```
> alphas := ["a", "b", "c", "d", "e", "f", "g"]:
```

With a range designation of 1..3, we obtain a sublist of the first three elements:

```
> alphas[1..3];
```

The following gives a sequence of these elements, which are not in a list:

```
> op(alphas[1..3]);
```

We can specify the fifth element of the list using the index 5, as follows:

```
> alphas[5];
```

Using this same notation in a sequence, *Maple* can return the **concatenation**, or joining, of parts of a list without changing the original list.

The command below returns a three-element list consisting of the fifth element of *alphas*, a list with the first three elements of *alphas*, and the last element of *alphas*, which has index -1. To have the values in a list, we enclose the sequence in brackets.

```
> [alphas[5], alphas[1..3], alphas[-1] ];
```

If we do not want the first three elements of *alphas* in brackets, we apply *op*, as follows, and obtain a five-element list:

```
> [alphas[5], op(alphas[1..3]), alphas[-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."

a. Using the 3-by-3 array *mat* below, return a list containing the last row. Thus,the returned list is [[7,8,9]].

```
> mat := [[1, 2, 3], [4, 5, 6], [7, 8, 9]]:
```

b. Write a command to return an array that is equal to *mat* except the last row of *mat* appears as the first and last rows of the new matrix. Thus, the new matrix has four rows, and its second, third and fourth rows are equal to the rows of *mat*. In your command, do not type specific numbers from *mat* but use *Maple* notation to take a sublist, join it to the sequence of elements in *mat*, and place the results in a list. Display the answer as a rectangular array of numbers.

c. Write a command to return a list with the last row of *mat*, all the rows of *mat*, and the first row of *mat*. Thus, for *mat* having three rows, the new matrix has five rows. Display the answer as a rectangular array of numbers.

Length

For generality in functions, it is sometimes useful to obtain a list's length, or the number of elements at the top level of the list. The following form using the *Maple* function ***nops*** returns the number of elements in *expr*:

nops(*expr*)

The length returned for each of the lists below is 3. In the second example, the elements are lists.

```
> nops(["a", "b", "c"]);
```

```
> nops([ [1, 2], [ 0.7, [[[8]]] ], [] ]);
```

Quick Review Question 2 Write a statement to assign to *lst* a list of all zeros that is of random length between 5 and 15 elements. Display the length of *lst*.

Grid Graphics

Frequently, simulations involve evolving rectangular arrays of numbers, and pictorial representations of these matrices can help scientists understand the information. For example, the segment below defines a procedure *val* that generates random integers 0, 1, or 2, and forms a 3-by-3 matrix, *mat02*, or list of $n = 3$ lists, each consisting of three such integers. The last statement in the segment displays *mat02* as a rectangular array.

```
> n := 3:
  val := rand(0..2):
  mat02 := [seq(
                [seq(val(), j = 1..n)],
             i = 1..n)]:
  convert(mat02, array);
```

The *Maple* function ***listdensityplot*** from the *plots* package represents such a matrix as a grid with a level of gray or color corresponding to the number in each cell. Thus, the graphics that the following command produces represents the minimum number, 0, as black, the maximum number, 2, as white, and the intermediate number, 1, is a shade of gray:

```
> plots[listdensityplot] (mat02);
```

Comparison of the array form of *mat02* and the picture reveals that the rows of numbers, read from left to right, correspond to the columns of squares, viewed from bottom to top. Thus, the display for the first row of the matrix occurs on the left, while the third row corresponds to the right column of the picture. For simulations, we often do not care about such ordering. However, if visualization of the data is aided by displaying the grid as the numbers appear in the matrix, the order of the rows can be reversed with the *Maple* command ***Reverse*** and the result transposed with *Transpose*. Both these functions are in the

package ***ListTools***.

In general, the following call in long form reverses the order of the elements in an expression, such as a list:

```
ListTools[Reverse]( expr)
```

The following segment returns the reverse of the list [1, 2, 3]:

```
> ListTools[Reverse] ([1,2,3]);
```

Reversal occurs at the top level. Thus, the reverse of a list of lists, [*l1*, *l2*, ..., *ln*], returns the list [*ln*, ..., *l2*, *l1*] without reversing the elements in the list, *l1*, *l2*, ..., and *ln*. Thus, the following reverse of [[1,2,3], [4, 5]] swaps the two elements, [1,2,3] and [4, 5], without reversing either:

```
> ListTools[Reverse]([ [1, 2, 3], [4, 5] ]);
```

Applying *listdensityplot* to the transpose of the reversed list, we display the graphics corresponding to the rows from top to bottom. With both functions in the *ListTools*, we include the package before calling the functions, as follows:

```
> with(ListTools):
  plots[listdensityplot](
   Transpose(Reverse(mat02))  );
```

Several options are useful with *listdensityplot*. Because the axes usually do not provide useful information in the visualization of a simulation, we eliminate them with the option-value pair ***axes = none***.

In simulations, we usually have a sequence of grids, each representing the result at one time step. However, a particular matrix corresponding to a grid may not contain all the possible values. For example, in simulating the spread of fire with 2 representing a burning cell, at a particular step, no fire might be burning so that 2 would not appear in the matrix. *Maple* determines the level of gray corresponding to each number depending on the range of numbers present. When performing animations of a sequence of grids, we want the same range of numbers applicable for each grid. Thus, we designate the range for *Maple* to use in each *listdensityplot* with the option-value pair ***range = n..m***.

With the option-value pair ***colorstyle = RGB***, we indicate that *listdensityplot* should employ colors instead of gray levels. For the above example, the following command indicates the graphics should employ colors, *range* = 0..2, and no axes:

```
> plots[listdensityplot](mat02,
   axes = none, range = 0..2  );
```

Quick Review Question 3

a. Generate a 10-by-10 array of random floating point numbers between 0 and 1, and visualize the array.

b. Generate a sequence of five 10-by-10 arrays of random floating point numbers between 0 and 1. Animate visualizations of the arrays. Use *listdensityplot*.

c. Generate a 10-by-10 array of floating point numbers varying from 0 to 0.9 in steps of 0.1 on each row. Display the corresponding graphics using *Transpose* and *Reverse* so that the display has a column of black on the left and a column of white on the right.

d. Smooth out the display with a 100-by-100 array and appropriately smaller step sizes.

The function *listdensityplot* is convenient for production of a "quick-and-dirty" visualization of a simulation. However, with this function, *Maple* employs default designations for the gray levels or colors. Frequently, an effective visualization employs colors that suggest meanings, such as red for fire and green for vegetation. To enable us to designate the color correspondence, we present an alternative approach using polygons. We start by defining a list with the coordinates in counterclockwise order of a unit square, *aSquare*, as follows:

```
> aSquare :=[[0, 0], [0, 1],[1, 1],[1, 0]]:
```

The function ***polygonplot*** from the *plots* package can display this polygon using the usual *plots* options, such as *color* in the following long form of the command:

```
> plots[polygonplot] (aSquare,
      color = red, axes = none);
```

The function ***translate*** from the ***plottools*** package translates a plot structure by designated amounts in the horizontal and vertical directions. The following segment defines a graphics, *redSquare*, for a red square and a graphics, *greenSquare*, for a green square. We display the squares together with *greenSquare* translated 2 units in the horizontal direction and 1 unit in the vertical direction. So that the squares continue to appear as squares and not elongated rectangles, we employ the *scaling* option with value *CONSTRAINED* .

```
> redSquare := plots[polygonplot](aSquare,
      color = red, axes = none):
  greenSquare := plots[polygonplot](aSquare,
      color = green, axes = none):
  plots[display]([redSquare,
   plottools[translate](greenSquare, 2, 1)  ],
      scaling = CONSTRAINED);
```

For visualizing an array, for each number, we construct a square with an appropriate color and translate this square into position. We illustrate the process with the following 4-by-3 array:

```
> matEx:=[[0, 0, 1], [0, 1, 0], [1, 0, 0], [1, 1, 0]]:
```

To specify the coloring, we define a function *matchColor* with parameters for an array, *mat*, and indices into that array, *i* and *j*. If $mat_{i,j}$ is 0, *matchColor* returns *yellow*; and if the array element is 1, the function returns forest green, which has RGB values of 0.1, 0.75, and 0.2, respectively.

```
> eval(matEx, [0 = yellow,
       1 = COLOR(RGB, 0.1, 0.75, 0.2)]);
```

```
> unassign(matchColor):
  matchColor := (mat, i, j)->eval(mat[i, j],
      [0 = yellow,
       1 = COLOR(RGB, 0.1, 0.75, 0.2)]):
```

For each row *i* and column *j*, we construct a graphics for a square using *polygonplot* with *matchColor* to designate the color corresponding to the array value at that location. Then, we translate the square into position by again using *i* and *j*. We show the resulting rectangular grid in yellow and forest green with *display*.

```
> plotGrid := seq(seq(
      plottools[translate](
          plots[polygonplot](aSquare,
              axes = none, scaling = CONSTRAINED,
      color = matchColor(matEx, i, j) ),
    j, i),
```

```
   j = 1..3), i = 1..4  ):
> plots[display](plotGrid);
```

If desirable, we can reverse the rows of the array to produce a graphics where the rows from top to bottom correspond to the array rows from top to bottom.

Quick Review Question 4

a. Generate a 10-by-10 array of random integers in {0, 1, 2}, and visualize the array with 0 as *pink*, 1 as *orange*, and 2 as *yellow*. Use *polygonplot*.

b. Generate a sequence of five 10-by-10 arrays of random integers in {0, 1, 2} . Animate visualizations of the arrays with the color designations as in Part a. Use *polygonplot*.

c. Generate a 10-by-10 array of floating point numbers varying from 0 to 0.9 in steps of 0.1 on each row. Display the corresponding graphics with corresponding increases in the amounts of red. Have zero green and blue components.

d. Smooth out the display with a 100-by-100 array and appropriately smaller step sizes.

Types

The **type** of a variable indicates the kind of data it can store. Some important types are as follows:

integer - integer data, or data in the set {..., -3, -2, -1, 0, 1, 2, 3, ...}

numeric - includes integer and floating point data. such as -38.2

character - single symbol in quotation marks, such as "b" or "+"

list - list data, such as [1,5,3]

anything - any valid expression except a sequence

positive - positive number

negative - negative number

nonnegative - numbers greater than or equal to zero

Function definitions and rules, which we cover shortly, sometimes need to specify the types of parameters. To indicate that a parameter, such as *L*, is of a certain type, such as *list*, we enter the type after the parameter name and double colons, such as in the function, *rest*, below. For an empty *list* argument, the function returns the argument; and for a nonempty list, the function returns the sublist with the first element omitted.

```
> rest := proc(L::list)
     `if`(L = [], [], L[2..-1]);
  end proc:
```

The following calls illustrate that we can apply the function to a *list* argument but not to a number.

```
> rest([]);
```

```
> rest([1,5,7,9,2]);
```

```
> rest(2);
```

Quick Review Question 5 Define a function, *h*, with one parameter, *x*, that only accepts integer arguments and returns x^2. Illustrate that the function returns the proper value for integer arguments 3, -3, and 0 but has no definition for arguments of other types, such as 0.3 and 1/2.

Matching Patterns for Definitions

Frequently, a *Maple* function that drives a simulation has different definitions for various configurations of arguments. To write a *Maple* program with several branches, we can employ nested calls to *if*. However, using several **rules** for a procedure, we can make the program easier to correct, modify, and understand. We can use the functions ***define*** and ***definemore*** for rule definitions and ***undefine*** to remove definitions associated with a operator name. In the following general form of *define*, we associate one or more rules with the operator *oper:*

define(***oper, rule1, rule2, ...rulen*** **)**

For example, for the operator *exFnc* below, when the argument is 5, the function returns 2. The second rule returns 3 times a numeric argument. The graph of this function is a straight line with a hole at $x = 5$ and the point (5, 2). As the calls to *exFnc* indicate, when we use any argument other than 5, the second definition applies; but with an argument of 5, *Maple* employs the first definition with its match of the pattern 5.

```
> undefine(exFnc);
```

```
> define(exFnc,
     exFnc(5) = 2,              # rule A
     exFnc(x::numeric) = 3*x    # rule B
  );
```

```
> exFnc(4);
```

```
> exFnc(5);
```

When a function has alternate definitions, *Maple* employs the first definition in which the arguments match the pattern of the parameters. For example, in the definition of *exFnc2* below, we reverse rules *A* and *B* from the above definition of *exFnc*. Because 5 matches the pattern *x::numeric*, *Maple* uses the first rule to evaluate *exFnc2*(5) as 3*5, or 15. Thus, the second rule never applies.

```
> undefine(exFnc2);
```

```
> define(exFnc2,
     exFnc2(x::numeric) = 3*x ,   # rule B
     exFnc2(5) = 2                # rule A NEVER APPLIED
  );
```

```
> exFnc2(5);
```

Sometimes in simulations, we have a number of situations to consider. Grouping rules into smaller units can make them more readable. We can use *define* for the first grouping and can employ *definemore* for subsequent groups. Again, the rules are applied in the order in which they appear.

```
> undefine(exFnc);
```

```
> define(exFnc,
     exFnc(5) = 2,              # rule A
     exFnc(6) = 2               # rule C
  );
```

```
> definemore(exFnc,
     exFnc(x::numeric) = 3*x    # rule B
  );
```

```
> exFnc(5);
```

```
> exFnc(6);
```

```
> exFnc(4);
```

As another example, in one kind of random walk, depending on the value of a random integer, the next point in a path might be the current site or one step in any north, east, south, or west direction. With *r* storing a random number, the nested *if* calls below return the next point in a random walk. This point is an ordered pair with no change or an increment or decrement by 1 of *x* or *y*, based on the value of *r*.

```
> rand0to3 := rand(0..3):
  r := rand0to3();
  x := 5:
  y := 5:
```

```
> `if`(r = 0, [ x + 1, y],        # east
    `if`(r = 1, [x - 1, y],       # west
      `if`(r = 2, [x, y + 1],     # north
        `if`(r = 3, [x, y - 1],   # south
                [x, y]))));       # site
```

The following rules for *dirFnc* generate the same logic. Subsequent calls to the function illustrate its action. Depending on the value of the first argument, the appropriate rule is invoked.

```
> undefine(dirFnc):
  define(dirFnc,
     dirFnc(0, x::numeric, y::numeric) = [ x + 1, y],
        # east
     dirFnc(1, x::numeric, y::numeric) = [ x - 1, y],
        # west
     dirFnc(2, x::numeric, y::numeric) = [ x, y + 1],
        # north
     dirFnc(3, x::numeric, y::numeric) = [ x, y - 1],
        # south
     dirFnc(r::numeric, x::numeric, y::numeric)
        = [x, y]   # site
  );
```

With the first argument being 2 in the following call to *dirFnc*, the "north" definition applies to return the point to the north of [17, 35]:

```
> dirFnc(2, 17, 35);
```

The following call to *dirFnc* with a first argument of 5 invokes the most general definition to return the site designated by the second and third arguments:

```
> dirFnc(5, 17, 35);
```

Without a type designation, such as *numeric* and *anything*, *Maple* would attempt to match the parameter symbol. For example, consider the following rule:

```
> define(matchx,
     matchx(x) = x^2
  );
```

In the following calls to *matchx*, we see that the rule only applies when the argument is exactly the symbol *x*.

```
> matchx(x);
```

```
> matchx(3);
```

```
> matchx(y);
```

The following definitions of *f* return the absolute value of nonzero arguments. Thus, *f* returns any positive argument. However, if an argument is less than zero, *f* returns the negative of the argument. As the last call to *f* reveals, the function is undefined at 0.

```
> undefine(f):
  define(f,
     f(x::positive) = x,
     f(x::negative) = -x
  );
```

```
> f(3);
```

```
> f(-3);
```

```
> f(0);
```

Quick Review Question 6 Define *g* using rules with alterative definitions, not *if*. The function has three parameters, *n*, *a*, and *b*. With a first argument of 1, if *a* or *b* is 3, the function returns *a* + 1. With a first argument of 2, if *a* or *b* is 4, the function returns *b* + 1. Otherwise, the function returns *a* + *b*. Test the function thoroughly.