8.1 Computational Toolbox—Tools of the Trade: *Maple* Tutorial 3

File: *MapleTutorial3.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* Tutorial 1" and " *Maple* Tutorial 2." Tutorial 3 prepares you to use *Maple* to complete projects for this chapter and is useful for material in subsequent chapters. The tutorial introduces the following functions and concepts: list-related operations and functions, such as *seq*, display as a matrix, and transpose; additional graphics features and options, such as *scaling*, plot range, *color*, and *linestyle*; *display*; curve fitting with *LeastSquares*; replacement rules with *eval*; reading from a file; and logarithms. The module also gives examples along with Quick Review Questions for you to do with *Maple*. Execute all input cells to view the results of the examples.

Sequences

Lists are essential to *Maple*, and we can employ the *seq* command in brackets to generate a list from repeated evaluation of an expression. The following form of the command with index *i* generates a sequence of elements that are evaluations of expr for *i* varying from *imin* to *imax*:

seq(*expr*, *i* = *imin*..*imax*)

For example, the command below generates a sequence of five zeros. In this case, the expression, 0, does not use the index, *i*, which ranges from 1 to 5, to cause the generation of five elements.

```
> seq(0, i = 1..5);
```

To create a list, we place the command in brackets, as follows:

```
> [seq(0, i = 1..5)];
```

The following command, which does employ the index *x* in the expression, generates a list of successive positive powers of 2:

```
> [seq(2^x, x = 1..6)];
```

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."

In an example of the section "Appending to a List" in " *Maple* Tutorial 2," we defined *g*(*x*) = 3 sqrt(

, initialized the list *gLst* to be empty, and then stored *g*(*i*) for the positive integers *i* less than 10 in *gLst*. Give a command with *seq* to create *gLst* without defining *g*.

Quick Review Question 2 In an example of a *for* loop from " *Maple* Tutorial 1," we initialized *dist* to be 0 and then seven times changed its value with the assignment *dist* := *dist* + 2.25 until *dist* became 15.75. Write a command to generate a list, *distLst*, containing these values from 0 through 15.75.

Do not use *dist*.

We can use *seq* to form a list of ordered pairs, too. For example, after defining $f(x) = x^2$, we generate a list of pairs of values of x and $f(x)$. To obtain the six values of x, 0.0, 0.5, 1.0, 1.5, 2.0, and 2.5, we have an index i going form 0 through 5 and have the x-coordinate be $0.5\,i$, as follows:

```
> unassign(f):
  f := x-> x^2;
  flst := [seq( [0.5*i, f(0.5*i)] , i = 0..5)];
```

Quick Review Question 3 In last Quick Review Question of "*Maple* Tutorial 2," you generated the following list (*lst*) of 30 points: [[0, 1.2], [0.25, 1.44], [0.5, 1.728], …, [7.25, 237.376]]. The first coordinate was 0.25 times an index *i* that varied from 0 to 29. The second coordinate, *y*, which was 1 before the loop, was 1.2 its previous value inside the loop. Thus, the second coordinate is a power of 1.2. Instead of using a *for* loop as in the previous tutorial, create the list (*lst*) using *seq*.

The command below generates a nested list that is comparable to a two dimensional array, where for each value of *i* from *imin* to *imax*, *j* varies from *jmin* to *jmax*. The result is a list that contains lists

```
seq( [seq( expr, j = jmin..jmax ) ],  i = imin..imax);
```

For example, the next command produces a list that is a 4-by-7 rectangular array:

```
> prod := [seq(
             [seq(x * y,  y = 3..9)],
           x = 1..4)];
```

At the highest level, *prod* contains four lists, one for each value of x from 1 through 4. Each of the four lists contains seven elements, one for each value of y from 3 through 9. At this lowest level, each element is the product of the corresponding values of x and y. Thus, the value of *prod* is a partial multiplication table.

Quick Review Question 4 Write a command to generate a table of values of (0.1*x)^(2*y - 1), where for each *x* value of 20, 21, 22, 23, 24, and 25, y takes on values 1, 2, and 3. Thus, the output is [[2., 8., 32.], [2.1, 9.261, 40.841], [2.2, 10.648, 51.5363], [2.3, 12.167, 64.3634], [2.4, 13.824, 79.6262], [2.5, 15.625, 97.6563]].

Displaying as a Matrix

We might find the information in list *flst* = [[0, 0], [0.5, 0.25], [1., 1.], [1.5, 2.25], [2., 4.], [2.5, 6.25]] above easier to understand if the format is in two columns instead of as one long list. The command ***convert*** can display an argument list of lists that have the same length as a rectangular ***array***, or matrix, of elements. The following command produces the list in two column output:

```
> convert(fLst, array);
```

Thus, we can think of this six-element list *flst* of ordered pairs as a rectangular array of six rows and two columns.

Quick Review Question 5 Write commands to display the following lists as rectangular arrays:
a. *lst* of 30 points from Quick Review Question 3 in the "Sequences" section

b. *prod* from the end of the "Sequences" section

c. The table from Quick Review Question 4. Copy the command that is the answer for that question to a new execution group as an argument for the *convert* command.

Transpose

In Module 8.3 on "Empirical Models," we deal with some examples where the first and second coordinates of data are in separate lists that need to be incorporated into one list of ordered pairs. For example, suppose for an hour a scientist measures amounts (in milligrams) of residues from a chemical reaction every 12 minutes, or 0.2 hours. The following command assigns to *tlst* the list of times, [0, 0.2, 0.4, 0.6, 0.8, 1.]:

```
> tlst := [seq(0.2*k, k = 0..5)];
```

The following *rlst* is a list of residue measurements:

```
> rlst := [0.00, 0.05, 0.16, 0.23, 0.55, 1.00];
```

The following expression produces on the output line a list of two lists, each with 6 elements:

```
> [tlst, rlst];
```

Using *convert*, we display this list as a rectangular array of two rows and six columns:

```
> convert([tlst, rlst], array);
```

The first row consists of the times, while the second stores the measured residues. To generate a list of ordered pairs of corresponding times and residues, we take the **transpose** of the list of lists [*tlst, rlst*]. Without changing the original list, the transpose function, ***Transpose***, in *Maple* package ***ListTools*** returns a list with the rows and columns swapped. Thus, after loading the package using the *with* command, a statement assigns the list of ordered pairs to *trans*:

```
> with(ListTools):
  trans := Transpose([tlst, rlst]);
```

Alternatively, we can call *Transpose* specifying its package without loading the entire package, as follows:

```
> trans := ListTools[Transpose]([tlst, rlst]);
```

The command *convert* to *array* shows the transposed list in six rows and two columns:

```
> convert(trans, array);
```

Definition The **transpose** of a matrix (rectangular array) is a matrix with the rows and columns exchanged from the original matrix.

Quick Review Question 6 Write a statement to generate a list *xLst* of positive integers from 1 through 9. From Quick Review Question 1, *gLst* stores the corresponding values of *g*(*x*). Write commands to assign to *pairsLst* the list of ordered pairs and to display *pairsLst* as a rectangular array.

Additional Graphics Options

In the previous *Maple* tutorials, we covered basic graphing of functions with *plot* and plotting data with *listplot*. In this tutorial, we discuss some additional features that are helpful in producing meaningful scientific visualizations.

The **aspect ratio** of a graphics is its width divided by its height. For example, if a graphics is 6 cm wide and 3 cm high, then its aspect ratio is 6 / 3 = 2; and if the dimensions are reversed, the aspect ratio is 3/ 6 = 1/2. *Maple* decides the aspect ratio that it considers best for each graph. To override the default and designate that a unit on the *x*-axis has the same length on the *y*-axis, we can employ the ***scaling*** option with value ***constrained*** as ***scaling = constrained***. Such specifications can be helpful for visualization of a graphics without distortion.

Definition The **aspect ratio** of a graphics is its width divided by its height.

Although we specify the part of the domain to graph, *Maple* decides on an appropriate part of the range unless we override the software's decision by specifying the plotting range as $y = c..d$. Thus, we specify the horizontal and vertical portions of the plot. The plot of $f(x) = x^2$ with domain from -3 to 3, range 0 to 9, the same proportions (*scaling* = *constrained*) on both axes, and the axes labeled (*labels*) as follows results in the graph shown:

```
> f := x->x^2:
  plot(f(x), x = -3..3,  y = 0..9,
     scaling = constrained ,
     labels = [" x ", " y "]);
```

Quick Review Question 7

a. Graph $y = \ln(x^2)$ from 0.1 to 6 without any options except *labels*.

b. Copy the command from Part a to a new execution group. In that cell, specify that the graph should be shown from 0 to 6 on the *x*-axis and from –2 to 3 on the *y*-axis.

c. Copy the command from Part b to a new cell. In that cell, specify that for the graph unit lengths should be the same on both axes. Observe the impact of each option on the graph.

Frequently, we wish to show more than one plot in the same graphics for comparisons. To do so, we group the functions in brackets in the *plot* command. The following segment defines *g* and plots *f* from above and g in the same graphics:

```
> f := x->x^2:
```

```
> unassign(g):
  g(x) := f(x) + 1;
  plot([f(x), g(x)] , x = -3..3);
```

In such situations, *Maple* uses different colors, such as red and green, to differentiate clearly between the two graphs. Instead of using the system's default colors, we can designate colors for display on a monitor or color printer, or we can employ assorted thicknesses or dashing for a black-and-white printer. With the option ***color***, we can indicate a predefined or a user-defined color for each function's graph with the option-value pair, ***color = value***. Some of the predefined colors are ***red, green, blue, black, yellow, orange, pink, purple, brown, tan, and grey*** . As with the functions, the two color values are grouped in brackets, as follows:

```
> plot([f(x), g(x)], x = -3..3,
     color = [purple, pink]  );
```

Sometimes, in a scientific visualization, we need to show a gradual change in color and consequently cannot use predefined color constants. However, we can designate a color in terms of its RGB value, which consists in three floating point numbers between 0 and 1 that indicate the amount of red, green, and blue in the composite color. We give the value using the ***COLOR*** structure with arguments ***RGB*** and three floating point numbers for the primary component amounts. As a result of the segment below, the first function, *f*, appears in a light cyan because its associated *RGB* color indicates no red (0), 50% (0.5) green, and full (1.0) blue. The second function, *g*, goes with the second color, which indicates a caramel color having RGB values of 1.0, 0.5, and 0.0, respectively.

```
> plot([f(x), g(x)], x = -3..3,
     color = [COLOR(RGB, 0.0, 0.5, 1.0),
              COLOR(RGB, 1.0, 0.5, 0.0)]
     );
```

Quick Review Question 8 Go to the input above and include the function $3x + 2$ in a shade of green

that has *RGB* values of 0.1, 0.9, and 0.1, respectively. Make the changes on the input; do not retype the cell.

To distinguish grays on a black-and-white printer, instead of using color, we can designate the ***thickness*** in terms of a nonnegative integer whose default is 0. The graphics displays the result of the following command with the function *g* thicker than *f* and both in black.

```
> plot([f(x), g(x)], x = -3..3,
     color = black, thickness = [0, 4]
     );
```

Quick Review Question 9 Copy the input from Quick Review Question 8 and paste it below. On the pasted execution group, have the graphs display in black-and-white with progressively thicker lines.

Alternatively, we can indicate a plot ***linestyle*** of ***DASH***, ***DOT***, ***DASHDOT***, or ***SOLID*** (the default) that indicates the graph is to be dashed, dotted, dash-dotted, or solid. The example below, has no extra specifications for the graph of *f* so that it appears as a solid line and indicates that the graph of *g* should be a thicker, dashed line.

```
> plot([f(x), g(x)], x = -3..3,
     color = black, thickness = [0, 4],
     linestyle = [SOLID, DASH]
     );
```

Quick Review Question 10 Copy the input from Quick Review Question 9 and paste it below. On the pasted execution group, have the graphs display in black-and-white. Display *f* having a thicker line that is dashed; display *g* with no extra options; and display 3 *x* + 2 dotted.

The **graphical user interface (GUI)** for *Maple* has an interactive option for graphics that leads the user through the plotting process and has cells for option values, buttons, and drop-down menus. To activate the plotting GUI, we load the ***plots*** package and call the plots function ***interactive*** with a function to be plotted as the argument. Alternatively, we do not fully load the package but call ***plots***[***interactive***] with a function argument. Both forms are below. After executing one of them, explore changing the plot options, observe the resulting graphics, and click the ***Command button*** to see the *Maple* command.

```
> with(plots):
  interactive(f(x));
```

```
> plots[interactive](f(x));
```

Showing Several Graphics Together

Several times in Module 8.3 on "Empirical Models," we need to display a plot of data points and a plot of a function in the same graphics. To do so, we generate each graphics, saving the results in variables, and then display the combination using the ***display*** command with the variables as arguments. For example, the segment below defines a list of points (*pts*), generates a graphics with the points, and stores the graphics in variable *lp*. As we see by executing the command, *Maple* does not display the graph.

```
> pts := [[-3, 20], [2, 20], [-1, -10], [4, 30], [0, 0]]:
  with(plots):
  lp := listplot(pts, labels = ["x", "y"],
     style = point, color = red, symbolsize = 20):
```

To see the graph, we employ the *display* function with argument being the variable *lp*, which obtained its value in the previous execution group. Because *display* is also in the *plots* package, we can invoke the function in either of the following ways

```
> display(lp);
```

```
> plots[display](lp);
```

The next segment plots a cubic polynomial, storing the graphics in variable *plt* and displaying the result:

```
> plt := plot(-3 + 9*x + 3*x^2 - x^3, x = -3..5,
     color = blue):
  plots[display](plt);
```

With the *display* command below, we show the data points and cubic together. The result illustrates that the cubic somewhat captures the trend of the data.

```
> plots[display](lp, plt);
```

Line Graphics Element

We can generate a line having the beginning and ending points, (*x1*, *y1*) and (*x2*, *y2*), by using the ***listplot*** command from the *plots* package. The first argument of *listplot* is a list containing the endpoints. We use the *style* default option of *line* with other desired plotting options. Thus, to display a thick line from point (1.309, 2.138) to point (1.68, 5.66), we employ the following command:

```
> listplot([[1.309, 2.138], [1.68, 5.66]],
     thickness = 5);
```

Quick Review Question 11 Replace each xxxxxx to complete the command below to display a line through points (0, 0) and (1, 3). Have graphics directives to designate a dashed line and RGB color levels of 0.4 for red, green, and blue. Use the full designation of the package and function name for plotting.

```
> xxxxxx[xxxxxx](xxxxxx[0,0],[1,3] xxxxxx,
     xxxxxx = xxxxxx,
     xxxxxx = xxxxxx(xxxxxx, 0.4, 0.4, 0.4));
```

Curve Fitting

In Module 8.3 on "Empirical Models," we investigate discovering functions that capture the trend of data. To do so, we employ the *Maple* function ***LeastSquares*** from the package ***CurveFitting***. Below is one form of loading the package and calling *LeastSquares* to fit an equation, *eqn*, in an independent variable, *var*, to the data. We can present the data in a list of ordered pairs, *pts*. We are requesting *Maple* perform the calculations so that the equation is the best least square fit to the data.

```
with(CurveFitting):
LeastSquares(pts, var, curve = eqn)
```

Maple returns the function that "best" fits the data in a list of points. For example, consider the list *pts* from above:

```
> pts := [[-3, 20], [2, 20], [-1, -10], [4, 30], [0, 0]]:
```

The following command loads the *CurveFitting* package and fits a cubic polynomial $a\,x^3 + b\,x^2 + c\,x + d$ to data:

```
> with(CurveFitting):
  fitCubic := LeastSquares(pts, x,
   curve = a*x^3 + b*x^2 + c*x + d );
```

We plot *fitCubic* as follows, storing the result in variable *cubicPlot*:

```
> cubicPlot := plot(fitCubic, x = -3..4,
     color = blue):
  plots[display](cubicPlot);
```

After assigning the data and cubic plots to *lp* and *cubicPlot*, respectively, we can show the two graphics together using *display*, as follows:

```
> lp := plots[listplot] (pts, labels = [" x ", " y "],
     style = point):
  plots[display](lp, cubicPlot);
```

An alternative form of *LeastSquares*, which follows, presents the data as a list of first coordinate data, *x_data_list*, and a list of second coordinate data, *y_data_list*:

```
with(CurveFitting):
LeastSquares(x_data_list, y_data_list, var, curve = eqn)
```

The above list of points, *pts*, consists of the following lists of *x*- and *y*-coordinates:

```
> xLst := [-3, 2, -1, 4, 0]:
  yLst := [20, 20, -10, 30, 0]:
```

Thus, with the *CurveFitting* package loaded, we can employ the following form of the *LeastSquares* command:

```
> fitCubic := LeastSquares( xLst, yLst, x,
      curve = a*x^3 + b*x^2 + c*x + d);
```

Quick Review Question 12

a. Plot the list of points stored in variable *pts2* below and save the graphics in variable *pts2Plot*. Have the points be green and a slightly larger size than those in the previous execution group.

```
> pts2 := [[0.4, 0.16], [0.6, 0.23], [0.8, 0.55], [1.0, 1.0]]:
```

b. Give the command to have *curveFit* store the equation of the line to fit "best" *pts2*. Recall that the general equation of a line is $y = mx + b$.

c. Plot the line from Part b and store the graphics in variable *curvePlot*.

d. Show the graphics for *pts2Plot* and *curvePlot* together.

e. Copy the command from Part b to a new execution group. Edit the copied cell to fit a quadratic, $y = a\,x^2 + b\,x + c$, to the data.

f. Copy the command from Part c to a new execution group and re-execute to plot the quadratic.

g. Copy the command from Part c to a new execution group and re-execute to show the data and quadratic in the same graphics.

LeastSquares gives the least-squares fit to a list of data. In Module 8.3 on "Empirical Models," we explain least-squares fit and its utility in working with data.

Mapping

In Module 8.3, as part of the process of capturing the trend of data, we need often need to perform an operation, such as squaring, on each element of a list. The *Maple* function ***map*** enables us to apply a procedure element-by-element to a list. One form of the function call, which follows, applies the procedure *fnc* to each element of the list *lst:*

map(*fnc*, *lst*)

For example, consider the following function, *sqr*, to square a number:

```
> sqr := x -> x^2:
```

Also, consider the following list, *xLst*, of numbers:

```
> xLst := [3, 5, -2]:
```

The following call to *map* applies *sqr* to each element of *xLst* and returns the list [*sqr*(3), *sqr*(5), *sqr*(-2)]:

```
> map(sqr, xLst);
```

Often, we do not need a procedure, such as *sqr*, elsewhere. In this case, we can employ a **functional operator**, which is like an unnamed procedure. With *sqr*, the functional operator consists of the procedure definition to the right of " *sqr* :=". We use this functional operator in place of *sqr* in the call to *map*, as follows:

```
> map(x -> x^2, xLst);
```

Quick Review Question 13 Consider the following lists of *x*- and *y*-coordinates from *pts2* of Quick Review Question 12:

```
> xLst2 := [0.4, 0.6, 0.8, 1.]:
  yLst2 := [0.16, 0.23, 0.55,  1.0]:
```

a. Using a functional operator, assign to *yLst2Power* a list with every element of *yLst2* raised to the 0.1 power.

b. Using *Transpose* from *ListTools*, assign to *pairsLst* a list of ordered pairs with corresponding elements from *xLst2* and *yLst2Power*.

c. Plot the points of *pairsLst* and compare the shape of the graph with *pts2Plot* of Quick Review Question 12 a.

d. Define a function, *tenthRoot*, to take the tenth root (raise to the 0.1 power) of an argument.

e. Using *map*, obtain a list with *tenthRoot* applied to each element of *yLst2*. Compare this answer with that of Part a.

Rules

Maple ***eval*** function enables us to have transformation rules, or to change expressions from one form to another. To replace every occurrence of *x* with *a* in expression *expr*, we employ the following form of the rule:

eval(*expr*, *x* = *a*)

For example, suppose *fitDataEq* is $0.0225362 + 0.75118\,z$, and we wish to replace z with $x^{3.9}$ in the expression *fitDataEq*. We use the *eval* command as in the following segment:

```
> fitDataEq := 0.0225362 + 0.75118*z:
  eval(fitData, z = x^3.9);
```

Moreover, we can define a function *f* equal to the result, as follows:

```
> unassign(f);
```

```
f := x->eval(fitData, z = x^3.9);
```

To make several substitutions, we have a list (with brackets) or set (with braces) of equations, *eqns*, for the rules, as follows:

eval(*expr*, *eqns*)

For example, we can replace z with $x^{3.9}$, a with 25, and b with 3 in an expression, as follows:

```
> eval(a + b*z, [z = x^3.9, a = 25, b = 3]);
```

Alternatively, we can place the substitution rules in a set using braces instead of a list, as follows:

```
> eval(a + b*z, {z = x^3.9, a = 25, b = 3});
```

Quick Review Question 14 Add rule replacements to the following line to return an expression replacing *u* with 2^x and *v* with ln(*y*).

```
> u + 7*v;
```

Reading from a File

Files can store huge amounts of data and simplify input. Links to data files for various projects appear on the textbook's website. For example, Module 8.3 on "Empirical Models" uses the file *DanWoodEMData.dat* , which follows, and several much larger files:

```
1.309  2.138
1.471  3.421
1.490  3.597
1.565  4.340
1.611  4.882
1.680  5.660
```

The *Maple* function ***readdata*** can read a file of numbers or data of other types into a table. One form of the command is as follows:

readdata["*filename*", *n*]

The file name must appear in quotation marks if the name contains characters other than letters of the alphabet or digits. Also, this name must be in the path of where *Maple* looks, or we must specify the full path name of the file. The second argument is a positive integer indicating the number of columns of data. *Maple* returns the data in a list with values on each line in a list. Thus, to have pairs of floating point numbers grouped into two-element lists, we can use a second argument of 2. For example, the file *DanWoodEMData.dat* consists of two columns of data for x- and y-values. To read the data into a list of points with the coordinates of a point in brackets and to store the resulting list in the variable *pts*, we employ the following command:

```
> pts := readdata("DanWoodEmData.dat", 2);
```

Quick Review Question 15 Replace each xxxxxx to complete the command to read the data of *DanWoodEMData.dat* into a list, *lst*, of 12 numbers:

```
> lst := xxxxxx  DanWoodEMData.dat  xxxxxx
```

Logarithms

See the section on "Logarithmic Functions" from Module 8.2 on "Function Tutorial" for a discussion of logarithms.

In *Maple*, the common logarithm of n is ***log10(n)***. Thus, the following call to the function returns 3:

```
> log10(1000)
```

The natural logarithm of n is *ln(n)* in *Maple*. Thus, ln(50.0) returns 3.91202 because $e^{3.91202}$ is 50. Bases other than e or 10 are permissible as long as the base is greater than 1. In general, *Maple's* ***log[b](n)*** is the logarithm to the base b of n.

Quick Review Question 16

a. Plot 2^x and ln x on the same graph.

b. Evaluate the common logarithm of 7 as a floating point number.