

## 8.1 *MATLAB* Tutorial 3

*Introduction to Computational Science: Modeling and Simulation for the Sciences*

Angela B. Shiflet and George W. Shiflet

Wofford College

© 2006 by Princeton University Press

### Introduction

We recommend that you work through the introduction with a copy of *MATLAB*, answering all Quick Review Questions. *MATLAB* is available from MathWorks, Inc. (<http://www.mathworks.com/>).

The prerequisites to this tutorial are "*MATLAB* Tutorial 1" and "*MATLAB* Tutorial 2." Tutorial 3 prepares you to use *MATLAB* for material in this and subsequent chapters. The tutorial introduces the following functions and concepts: array operations, transpose; additional graphics options, such as for equal axes, plotting area, line width, and line style; fitting; rule substitutions; reading data files; and logarithms.

### Array Operations

Arrays are essential to *MATLAB*, and we can perform operations on entire arrays. To perform addition, subtraction, multiplication, or division of a scalar (number) by every element in an array, we use the usual operator of +, -, \*, or /, respectively. For example, consider the following vector:

```
vec = [4:0.3:5.2]
vec =
    4.0000    4.3000    4.6000    4.9000    5.2000
```

The following input statements (in red) perform the same operation on every element of *vec*, returning the appropriate output array (in blue) without changing the value of *vec*:

```
vec + 10
14.0000    14.3000    14.6000    14.9000    15.2000

vec - 5
-1.0000   -0.7000   -0.4000   -0.1000    0.2000

vec * 10
40    43    46    49    52

vec / 10
0.4000    0.4300    0.4600    0.4900    0.5200
```

To change the value of *vec*, we must assign the result of an operation to *vec*, such as follows:

```
vec = vec * 10
vec =
```

40    43    46    49    52

As the following Quick Review Question illustrates, we can also apply other functions, such as the square root function (*sqrt*), that operate on a single number to every element of an array.

**Quick Review Question 1** Start a new *M*-File. In opening comments, have "MATLAB Tutorial 3 Answers" and your name. Save the file under the name *MATLABTutorial3Ans.m*. In the file, preface this and all subsequent Quick Review Questions with a comment that has "QRQ" and the question number, such as follows:

```
% QRQ 1 a
```

- a. With one assignment statement, make *ar* a 2-by-4 array of all zeros.
- b. With one assignment statement, make the first row of *ar* be the sequence of positive integers 1, 3, 5, 7.
- c. Return the product of 3 by every element of *ar* without changing *ar*.
- d. Return the square root of every element of *ar* without changing *ar*.
- e. Add 2 to every element of *ar*, changing the value of *ar*.

If two arrays have the same dimensions, we can perform operations that combine corresponding elements. Addition and subtraction use the typical operators of + and -, respectively. However, the **array multiplication, division, and exponentiation operators**, with the operations performed element-by-element on corresponding elements, include a period to be *.\**, *./*, and *.^*, respectively. The following statements illustrate these operations using *vec* and another five-element vector:

```
vec2 = [3  -1  0  8  1];
vec .* vec2
120   -43    0   392   52

vec2 ./ vec
0.0750  -0.0233    0   0.1633  0.0192

vec2 .^ 2
9     1     0   64     1
```

Besides performing the same operation on every element of an array, some functions combine the values in an array. For example, the function *sum* returns the sum of all the elements in an array. Thus, with *vec2* being [3, -1, 0, 8, 1], the following command returns  $3 + -1 + 9 + 8 + 1 = 11$ :

```
sum(vec2)
```

In the case of a two-dimensional array, *sum* returns a vector with the sum of each column. Thus, to obtain the sum of an entire two-dimensional array, we can take the *sum* of the *sum* of the array.

**Quick Review Question 2**

- a. With one assignment statement, make *br* a 2-by-4 array of all zeros.
- b. With one assignment statement, make the first column of *br* contain ones.
- c. Make the first row, third element of *br* be 5.
- d. Display *ar*.
- e. Without changing either array, obtain the sum of *ar* and *br*.
- f. Return an array where corresponding elements of *ar* and *br* are multiplied.
- g. Define a function *sqr* to square a parameter. The function should work for numeric and array arguments.
- h. Using *sqr* from Part g, return an array with every element of *ar* squared.
- i. Give a command to return the sum of the squares of the elements of *ar*.

**Transpose**

In Module 8.3 on "Empirical Models," we deal with some examples where the first and second coordinates of data points are in separate lists that need be in one array of ordered pairs with one pair per row. 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 = [0:0.2:1];
```

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 on the input line produces output of a rectangular array of two rows and six columns:

```
[tlst; rlst]
ans =
    Columns 1 through 5
         0    0.2000    0.4000    0.6000    0.8000
         0    0.0500    0.1600    0.2300    0.5500
    Column 6
         1.0000
         1.0000
```

Thus, the array is as follows:

0.0	0.2	0.4	0.6	0.8	1.0
0.00	0.05	0.16	0.23	0.55	1.00

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 *[tlst; rlst]*. Without changing the original array, the transpose, which

*MATLAB* indicates with a trailing **apostrophe** (`'`), returns an array with the rows and columns swapped. Thus, the following statement assigns the list of ordered pairs to *trans*:

```
trans = [t1st; r1st]'
```

The output value of *trans* is as follows:

```

      0      0
0.2000  0.0500
0.4000  0.1600
0.6000  0.2300
0.8000  0.5500
1.0000  1.0000
```

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

**Quick Review Question 3** Write a statement to generate a list *xLst* of *x*-values, which are positive integers from 1 through 9. Using one assignment statement, have *gLst* store the corresponding values of  $3\sqrt{x}$ . Write commands to assign to *pairsLst* the array of ordered pairs with one ordered pair per row.

### Additional Graphics Options

In "*MATLAB* Tutorial 1," we covered basic graphing of a function with *plot*. 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  $3/6 = 1/2$ . *MATLAB* decides the aspect ratio that it considers best for each graph. To designate that a unit length on one axis is the same as on the other, we employ the option **axis equal**. Such specifications can be helpful for visualization of a graphics without distortion. To revert to *MATLAB*'s choice of aspect ratio, we designate **axis auto**.

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

Although we specify the part of the domain to graph, *MATLAB* decides on an appropriate part of the range unless we override the software's decision with another property, **axis**. The designation **axis([a b c d])** specifies the horizontal portion as being from *a* to *b* and the vertical portion from *c* to *d*. The following plot of  $f(x) = x^2$  with domain from -2 to 3, range from 0 to 9 (**axis([-2 3 0 9])**), the same proportions (**axis equal**) on both axes, and the axes labeled (*xlabel* and *ylabel*) results in the graph of Figure 1:

```
clear('f');
f = @(x) x .* x;

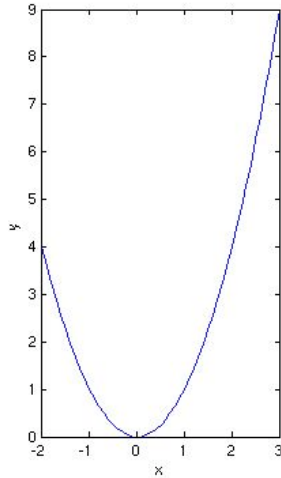
x = -2:0.1:3;
plot(x, f(x))
```

```

axis equal
axis([-2 3 0 9])
xlabel('x')
ylabel('y')

```

**Figure 1** Graph of  $x^2$  with equal units on each axis and specified plot region



#### Quick Review Question 4

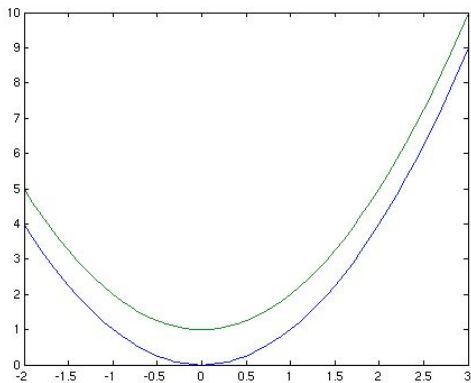
- Graph  $y = \ln(x^2)$  from 0.1 to 6 without any properties except axes labels.
- Specify that the graph should be shown from 0 to 6 on the  $x$ -axis and from  $-2$  to 4 on the  $y$ -axis.
- Specify that a unit on one axis should be the same length as a unit on the other axis. Observe the impact of each option on the graph.

Frequently for comparisons, we wish to show more than one plot in the same figure. To do so, we can alternate sequences of values independent and dependent variables as arguments in the `plot` command. The following segment defines  $g$  and plots  $f$  and  $g$  in the same graphics (Figure 2):

```

clear('g');
g = @(x) f(x) + 1;
plot(x, f(x), x, g(x))

```

**Figure 2** Graph of  $f(x) = x^2$  and  $g(x) = f(x) + 1$ 

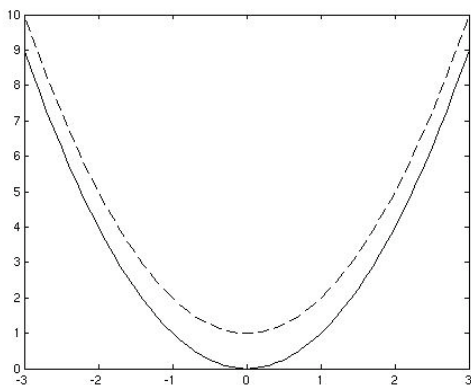
In such situations, to differentiate clearly between the two graphs, *MATLAB* shows them with different colors for display on a monitor or color printer. Alternatively, we can employ assorted thicknesses or dashing for a black-and-white printer. After the sequences for each graph, we can indicate the color in a string, such as 'k' for black in the following

```
x = -3:0.1:3;
plot(x, f(x), 'k', x, g(x), 'k')
```

**Quick Review Question 5** For the above plot, include the function  $3x + 2$  in green.

To distinguish graphs on a black-and-white printer, we can indicate a color of black with the string 'k' and in the same string a style of **dashed** with '--', **dotted** with '.', or **dash-dotted** with '-.'. The following example in black-and-white, which Figure 3 pictures, has no extra specifications for the graph of  $f$  so that it appears as a solid curve and indicates that the graph of  $g$  should be dashed.

```
plot(x, f(x), 'k', x, g(x), '--k')
```

**Figure 3** Plot from Figure 2 with  $g$  being dashed

**Quick Review Question 6** For the functions  $f$ ,  $g$ , and  $3x + 2$  from Quick Review Question 5, have the graphs display on one figure in black-and-white with  $f$  being dashed,  $g$  being dotted, and  $3x + 2$  being dash-dotted.

Alternatively, we can designate the thickness using the property '*LineWidth*'. The value is in number of **points**, where 1 point = 1/72 inch and the default is 0.5 points. The following command indicates to plot  $f$  with a line of thickness value 3, which is 3/72 inch = 1/24 inch:

```
plot(x, f(x), 'k', 'LineWidth', 3)
```

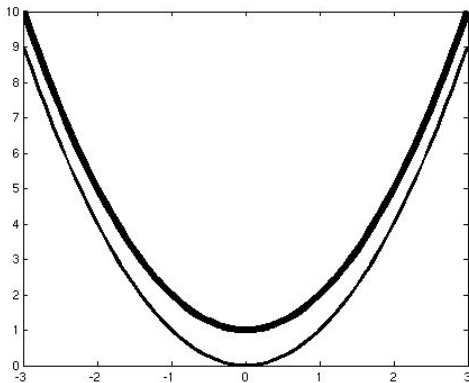
However, a *plot* command designates the line width for the entire figure, not just a graph of one function. In the next section, we indicate how to have separate line widths for different functions.

### Showing Several Graphics Together

To have two functions on the same graph while specifying properties, such as *LineWidth*, we graph one function and issue the command ***hold on***, which adds subsequent plots to the figure instead of replacing them. Additions continue until execution of the command ***hold off***. Figure 4 displays the final result of the following command sequence with the function  $g$  thicker than  $f$  and both in black:

```
plot(x, f(x), 'k', 'LineWidth', 3)
hold on
plot(x, g(x), 'k', 'LineWidth', 6)
hold off
```

**Figure 4** Plot from Figure 2 with  $g$  thicker than  $f$



**Quick Review Question 7** For the functions  $f$ ,  $g$ , and  $3x + 2$  from Quick Review Question 5, have the graphs display on one figure in black-and-white with progressively thicker lines.

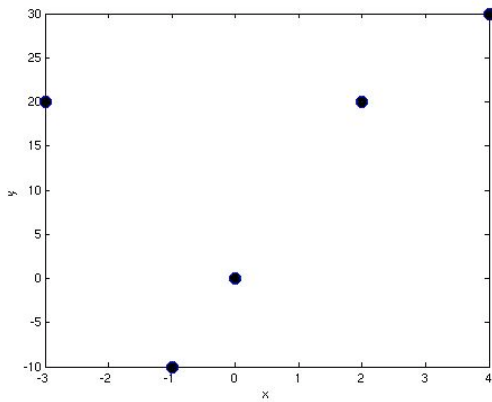
Similarly, several times in the module on "Empirical Models," we need to display a *plot* of data points and a *plot* of a function in the same graphics. To do so, we generate the first graphics, indicate *hold on*, and then display the second graphics. For example, the following segment defines a list of point coordinates, displays the points with *plot* as in Figure 5a, and indicates *hold on*:

```
xCoords = [-3, 2, -1, 4, 0];
yCoords = [20, 20, -10, 30, 0];
plot(xCoords, yCoords, 'o', 'MarkerSize', 10, 'MarkerFaceColor', 'k')
xlabel('x')
ylabel('y')
hold on
```

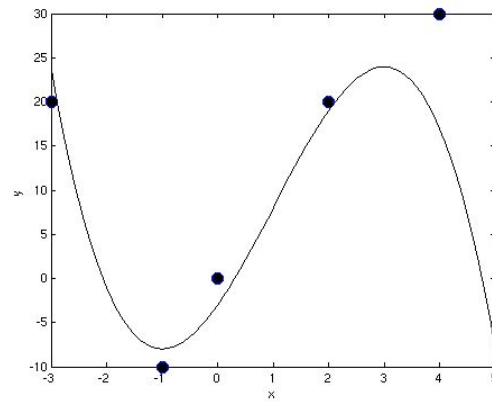
The next segment plots a cubic polynomial in the same figure (Figure 5b), illustrating that the cubic somewhat captures the trend of the data:

```
x = -3:0.1:5;
plot(x, -3 + 9*x + 3*x.*x - x.^3, 'k')
hold off
```

**Figure 5a** *plot* of data points



**Figure 5b** *plot* of cubic with Figure 5a



### Fit

In Module 8.3 on "Empirical Models," we investigate discovering functions that capture the trend of data. To do so, we employ the *MATLAB* functions *polyfit* and *polyval*. The form of a call to *polyfit* is as follows:

```
polyfit(x-coordinates, y-coordinates, n)
```

For nonnegative integer  $n$ , *MATLAB* returns coefficients  $(a_n, \dots, a_1, a_0)$  of the  $n$ -th degree polynomial  $a_n x^n + \dots + a_1 x + a_0$  that "best" fits the data points with coordinates in *x-coordinates* and *y-coordinates*. For example, the following command fits a cubic polynomial to the set of points  $\{(-3, 20), (2, 20), (-1, -10), (4, 30), (0, 0)\}$  above:

```
xCoords = [-3    2    -1    4    0];
yCoords = [20    20   -10    30    0];
fitCubic = polyfit(xCoords, yCoords, 3)
```

*MATLAB* returns the following list (*fitCubic*) of cubic polynomial coefficients:

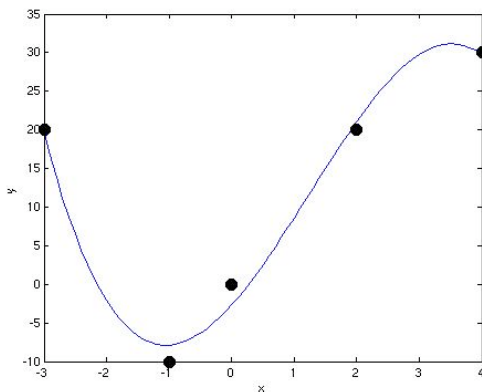
```
-0.8205    3.0439    9.0617   -2.6682
```

Thus, the cubic that "best" fits the data is  $-0.8205x^3 + 3.0439x^2 + 9.0617x - 2.6682$ . The *MATLAB* function *polyval* evaluates such a polynomial at an array of values. In the following segment, we assign to *x* a sequence of values; evaluate the polynomial with coefficients in *fitCubic* at these values, storing the results in *polyValues*; and plot the cubic through points with *x*-coordinates in *x* and *y*-coordinates in *polyValues*:

```
x = -3:0.1:4;
polyValues = polyval(fitCubic, x);
plot(x, polyValues)
```

After plotting these points, we show together the graphics of the points (Figure 5a) and the cubic to obtain Figure 6.

**Figure 6** Graphics of data points and fitted cubic



### Quick Review Question 8

- Plot the points in the set  $\{(0.4, 0.16), (0.6, 0.23), (0.8, 0.55), (1.0, 1.0)\}$ . Have the points be green with size 8.
- Give the command to have *curveFit* store the coefficients of the line to fit "best" the data. Recall that the general equation of a line is  $mx + b$ , which is a polynomial of degree 1.
- Plot the line from Part b on the same figure as that of Part a.
- Fit a quadratic,  $a_2x^2 + a_1x + a_0$ , to the data.
- Add a plot of this quadratic to the figure.
- No longer have *hold on* active.

The function *polyfit* 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.

## Rules

*MATLAB*'s transformation rules or substitutions enable us to change expressions from one form to another. The function *subs*, which has the following general form, substitutes *value* for *term* every place *term* occurs in *expr*:

```
subs(expr, term, value)
```

For example, suppose  $z$  and  $x$  are symbols, and *fitData* is  $23 + 751 * z$ . The following sequence replaces  $z$  with  $x^{3.9}$  in the expression *fitData*:

```
syms z x
subs(fitData, z, x.^3.9)
```

To define a function  $f$  equal to the result, we use the following command:

```
f = @(x) subs(23 + 751 * z, z, x.^3.9)
```

Thus, we are defining  $f(x)$  as  $23 + 751x^{3.9}$ . We can plot the function from 0 to 4 as follows:

```
vals = 0:0.1:4;
plot(vals, f(vals))
```

**Quick Review Question 9** Declare  $x$ ,  $y$ ,  $u$ , and  $v$  to be symbols. Then, perform two substitutions to return an expression replacing  $u$  with  $2^x$  and  $v$  with  $\ln(y)$ .

```
u + 7*v
```

Remember that the natural logarithm function in *MATLAB* is *log*.

## 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 *DanWoodEM.dat*, which appears in Table 1, and several much larger files, all with rectangular arrays of tab-delimited data.

**Table 1** *DanWoodEM.dat*

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 *MATLAB* function *load* can read such a file of numbers into an array. One form of the command is as follows:

```
load('FileName.dat')
```

The file name appears in apostrophes with an extension of *.dat* or *.txt*. Unless assigned to a variable, the data is stored in a variable with the name of the file, *FileName*.

For example, the file *DanWoodEM.dat* consists of two columns of data for *x*- and *y*-values. To read the data into an array of points with the coordinates of each point on a line and to store the resulting table in the variable *pts*, we employ the following command:

```
pts = load('DanWoodEM.dat')
```

The command returns the following array:

```
pts =
    1.3090    2.1380
    1.4710    3.4210
    1.4900    3.5970
    1.5650    4.3400
    1.6110    4.8820
    1.6800    5.6600
```

The data file must be in the path of where *MATLAB* looks, or we must specify the full path name of the file. We can add a path to the list of *MATLAB* paths from the *File* menu by selecting *Set Path...*, clicking *Add Folder...*, and navigating to the appropriate directory.

As an alternative to setting the path and using *load*, from the *File* menu we select *Import Data...* The *Import Wizard* displays the data and leads us through importing the data. For tab delimited numeric data in a rectangular array configuration, we press *Next* and then *Finish*.

**Quick Review Question 10** Read the data of *BoxBODEM.dat*, which Module 8.3 on "Empirical Models" uses, into an array, *lst*.

## Logarithms

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

In *MATLAB*, 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 **log(n)** in *MATLAB*. Thus, *log(50.0)* returns 3.9120 because  $e^{3.9120}$  is 50.0.

**Quick Review Question 11**

- Plot  $e^x$  and  $\ln x$  on the same graph.
- Evaluate  $\log_{10} 7$ .