

5.1 *MATLAB* Tutorial 2

Introduction to Computational Science: Modeling and Simulation for the Sciences

Angela B. Shiflet and George W. Shiflet

Wofford College

© 2010 by Princeton University Press

Introduction

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

The prerequisite to this tutorial is "*MATLAB* Tutorial 1." Tutorial 2 prepares you to use *MATLAB* to complete projects for this and subsequent chapters. The tutorial introduces the following concepts: arrays, plotting points, and appending. Enter and execute all input cells to review the results of the examples.

Arrays

Many *MATLAB* applications involve **arrays**, or rectangular configurations of numbers, expressions, or other items. **Brackets**, [], enclose the values in an array, and commas and/or blanks separate the values in a row, such as either of the following:

```
numList = [13, 36, 92]
```

or

```
numList = [13 36 92]
```

Execution of this assignment results in the following output:

```
numList =  
    13    36    92
```

We also employ an array to store an ordered pair, such as the following representation of the point (-3, 46):

```
orderedPair = [-3, 46]
```

Quick Review Question 1 Start a new *M*-File. In opening comments, have "MATLAB Tutorial 2 Answers" and your name. Save the file under the name *MATLABTutorial2Ans.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
```

Assign to *xLst* an array containing the numbers -3, 5, and 1.

Arrays, such as *numList* and *orderedPair*, consisting of one row (or of one column) are also called **vectors**. The elements of a vector have a numeric order starting with 1, and we can refer to a particular element by using the name of the list and parentheses, (), surrounding that number. For example, the second element of *numList* above is as follows:

```
numList(2)
```

Besides using a number, such as 2, we can employ a variable, such as *i*, that has a value, so that *numList(i)* refers to the *i*th element of the vector. Consequently, a *for* loop can process the elements of an array individually by having the varying loop index specify a list element.

Quick Review Question 2 Using *disp* and a *for* loop, print on separate lines the elements of *xLst* from Quick Review Question 1.

While commas and/or blanks separate elements in a row of an array, **semicolons (;)** separate rows. Thus, the following array of numbers represents a matrix with two rows and four columns of numbers:

```
mat = [45, 99, 203, -29; 775, 31, -582, 62]
```

or

```
mat = [45 99 203 -29; 775 31 -582 62]
```

Output from either of these statements is as follows:

```
mat =
    45    99   203   -29
   775    31  -582    62
```

To obtain the element in the first row and third column, 203, we employ the following notation that gives the row and column numbers and separates these indices with a comma:

```
mat(1, 3)
```

We can change the value of this element to, say, 54 by using an assignment statement, such as follows:

```
mat(1, 3) = 54
```

We reference the second row by giving 2 for the row index and the sequence of indices, 1:4, for the four columns, as follows:

```
mat(2, 1:4)
```

MATLAB returns the following:

```
ans =
```

```
775    31  -582    62
```

The last row or column is represented by *end*. Thus, another way to refer to the second row is to indicate the last row from its first to its last column, as follows:

```
mat(end, 1:end)
```

We reference columns, such as the third, in the same manner using a sequence in the row-index position, such as follows:

```
mat(1:end, 3)
```

An even easier way to reference an entire row or column is to use a **colon** (:) for the corresponding sequence such as follows for the last row and third column:

```
mat(end, : ) % last row
mat( : , 3)  % column 3
```

Quick Review Question 3

- Store in *ptLst* the following ordered pairs (points) with the first coordinates in the first column and the second coordinates in the second column: (-3, 6), (5, 2), and (1, 12)

Write commands to return the following parts of *ptLst*:

- The coordinates of the third point, (1, 12)
- The first coordinate of the second point, 5
- The second coordinate of the first point, 6
- The first row
- The second column

Plotting Points

To plot points in an array, we use *plot* with a string indicating the point style, such as 'o' for a **circle marker**. The following segment assigns an array of *x* coordinates to the variable *xCoords*, an array of corresponding *y* coordinates to *yCoords*, and then displays the points as small circles with the appropriate axes labels:

```
xCoords = [-3    2    -1    4    0];
yCoords = [ 2    2    -1    3    0];
plot(xCoords, yCoords, 'o')
xlabel('x')
ylabel('y')
```

Quick Review Question 4 Assign to *yLst* the array with numbers 6, 2, and 12. Graph the points with *x* coordinates in *xLst* from Quick Review Question 1 and corresponding *y* coordinates in *yLst*.

To have larger points, as in the command below, we employ the **property** name '*MarkerSize*', which is a character-string argument, with associated property value, which is a positive integer value (here 10). The default value is 6.

```
plot(xCoords, yCoords, 'o', 'MarkerSize', 10)
```

Another property indicates '*MarkerFaceColor*', which is the interior color of the marker, such as the circle marker. The property value can be a string with a character representing a color. Table 1 lists several color alternatives. The following command plots the above points with filled red circles:

```
plot(xCoords, yCoords, 'o', 'MarkerSize', 10, 'MarkerFaceColor', 'r')
```

Table 1 Some defined colors

Color	String
black	'k'
blue	'b'
cyan	'c'
green	'g'
magenta	'm'
red	'r'
white	'w'
yellow	'y'

Quick Review Question 5 Using the up arrow, recall your answer for Quick Review Question 4 to plot the points with coordinates in *xLst* and *yLst*. Adjust the command to have the points appear as larger circles filled with black.

Lines Connecting Points

Sometimes, it is helpful to visualize the path of an entity, such as an animal or a molecule, whose movement we are simulating. To generate the path, in the same string for indicating the marker format, we specify the format of connecting line segments. A **dash** (-) in single quotes, such as '-' or '-o', indicates solid line segments or solid line segments with circle points, respectively. The subsequent graph displays line segments joining pairs of adjacent points.

For example, suppose [-1, 0, -1, 0, 1, 2, 3, 2, 1, 0] is an array (*yValues*) of ten *y* values, where each element is randomly one more or less than the previous element. Considering each *y* value to occur at sequential ticks of the clock, we can draw a line graph to display the trend of the *y* values over time. Just plotting *yValues* causes the first coordinates of the points to be 1, 2, ..., 10 and the corresponding second coordinates to be from *yValues*. The segment to display the trend of *y* over time follows:

```
yValues = [-1, 0, -1, 0, 1, 2, 3, 2, 1, 0];
lp = plot(yValues, '-o')
```

Quick Review Question 6 Plot the list of points with coordinates in $xLst$ and $yLst$ from Quick Review Question 4. Label the axes, have the points be filled blue circles, and connect the points with line segments.

Appending

In simulations with *MATLAB*, we frequently build an array one element at a time. To do so, we can assign a value to the element beyond the end of the array. The form of the assignment statement, where $expr$ is an array and $elem$ is an additional element, follows:

$$expr(\mathbf{end} + 1) = elem$$

The following segment returns the value of the row array lst with a new element, 15, on the end:

```
lst = [11, 12, 13, 14];
lst(5) = 15
```

Output is as follows:

```
lst =
    11    12    13    14    15
```

To avoid counting the number of elements in the array, we can refer to the element with index $end + 1$, such as follows:

```
lst(end + 1) = 16
```

We can append a row or column onto the end of an array by specifying an appropriate range of elements. For example, consider mat , an array of two rows and four columns:

```
mat = [45 99 203 -29; 775 31 -582 62]
```

After execution of the following assignment statement, mat contains a third row of all fives:

```
mat(3, :) = [5, 5, 5, 5]
```

To append a column with values 1, 2, and 3, we give the range of row elements ($1:end$), the index of the new column ($end + 1$), and indicate that each value is on a new row by using semicolon separators:

```
mat(:, end + 1) = [1; 2; 3]
```

The resulting array is as follows:

```
mat =
    45    99    203   -29     1
   775    31   -582     62     2
     5     5     5     5     3
```

In a programming language, such as C, C++, or Java, when using a loop to count, we must initialize the counter to be zero before the loop. Similarly, we must initialize an array before building. When building an array from scratch, the initial value is usually an **empty list**, []. The segment below stores $3\sqrt{i}$ for the integers i from 1 through 6 in the list $gLst$, which is originally empty. Finally, we have *MATLAB* return the value of $gLst$.

```
gLst = [];
for i = 1:6
    gLst(end + 1) = 3 * sqrt(i);
end

gLst
```

The resulting array is [3.0000 4.2426 5.1962 6.0000 6.7082 7.3485].

Alternatively, if we know the final size in advance, we should initialize the $gLst$ as an array of zeros with that size. Such an initialization sets aside contiguous memory space and speeds computation. The designation for a **zero array** of n rows and m columns is **zeros(n, m)**. Thus, the following segment initializes a row of 6 zeros and changes the values of the elements appropriately:

```
gLst = zeros(1, 6);
for i = 1:6
    gLst(i) = 3 * sqrt(i);
end

gLst
```

Quick Review Question 7 Using the comments in the cell below as a guide, write a *MATLAB* segment to generate an array, $xValues$, of 30 x -coordinates and an array, $yValues$, of 30 y -coordinates. Initialize $xValues$ and $yValues$ to be 1-by-30 arrays of zeros. Initialize y to be 1. Inside the body of a *for* loop with an integer index i that goes from 1 through 30, make x be 0.25 times $(i - 1)$; make y be 1.2 times its previous value; and insert x and y as the i -th elements in the arrays $xValues$ and $yValues$, respectively. After creation of the arrays of coordinates, plot the corresponding points with line segments joining adjacent points. Have labeled axes.

```
% initialize array xValues to be a 1-by-30 array of zeros
% initialize array yValues to be a 1-by-30 array of zeros
% initialize y to be 1
% for i going from 1 through 30 in a loop do the following:
%   assign to x the expression 0.25 times (i - 1)
%   place x in the i-th column of xValues
%   assign to y the expression 1.2 times previous value of y
%   place y in the i-th column of yValues
%
% plot the points
```