

10.1 *MATLAB* Tutorial 5

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 this tutorial 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* Tutorials 1-4. Tutorial 5 prepares you to use *MATLAB* for material in this and subsequent chapters. The tutorial introduces the following functions and concepts: maximum, minimum, animation, function M-files, and several features for projects, including the *while* loop, logical operators, and testing membership.

Maximum and Minimum

The *MATLAB* function *max* returns the maximum of a vector argument. Thus, the 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 a two-dimensional array of numbers as argument:

```
max([x11, x12, ..., x1m; x21, x22, ..., x2m; ...; xn1, xn2, ..., xnm])
```

The result is a vector of the maximum in each column. The segment below generates a 3-by-3 array of numbers and returns the maximum values (3, 2, 5) in the columns:

```
lst = [-1, 2, -1; 0, 1, 5; 3, 2, 1];  
max(lst)
```

To obtain the maximum (5) in the entire two-dimensional array, we take the maximum of this value, or the maximum of the maximum, as follows:

```
max(max(lst))
```

In a similar fashion, the *MATLAB* function *min* returns the minimum of a vector of numeric arguments; or in the case of a two-dimensional array, *min* returns a vector a minimum column values.

Quick Review Question 1 Start a new *Script*. In opening comments, have "MATLAB Tutorial 5 Answers" and your name. Save the file under the name *MATLABTutorial5Ans.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
```

Write a segment to generate a list, *lst*, of 100 random floating point numbers between 0 and 1 and to display the maximum and minimum in the list.

Animation

One interesting use of a *for* loop is to draw several plots for an animation. In making the graphics, plots should have the same *axis* so that axes appear fixed during the animation. The segment below generates a sequence of plots of $0.5 \sin(x)$, $\sin(x)$, $1.5 \sin(x)$, ..., $5.0 \sin(x)$ with x varying from 0 to 2π to illustrate the impact the coefficient has on amplitude. However, for an effective demonstration, each graph must have the same display area, here from -5 to 5 on the y-axis.

```
x = 0:0.1:2*pi;
for i = 1:10
    y = 0.5 * i * sin(x);
    plot(x,y, '-');
    axis([0 2*pi -5 5]);
end
```

However, the animation can be fairly slow and jerky as *MATLAB* generates the animation one frame at a time.

Alternatively, we can store the frames in a vector and later replay the sequence as a movie. Immediately after generating the i -th frame of the animation, we save the graph in a vector using *getframe*, as follows

```
M(i) = getframe;
```

After creating and storing all the frames, we show the animation with the *movie* command, which has the vector, here *M*, and the number of times to show the movie as arguments. Thus, the following command displays the animation 10 times:

```
movie(M, 10)
```

The adjusted sequence to generate and play a movie five times follows:

```
x = 0:0.1:2*pi;
for i = 1:10
```

```

y = 0.5 * i * sin(x);
plot(x,y,'-');
axis([0 2*pi -5, 5]);
M(i) = getframe;
end
movie(M, 5)

```

Quick Review Question 2 The first two statements of following segment generate 10 random x - and y -coordinates between 0 and 1. We then plot the points as fairly large blue dots in a 1-by-1 rectangle. Adjust the code to generate 10 plots. The first graphics only displays the first point; the second graphics shows the first two points; and so forth. After execution, animate to show the points appearing one at a time. Why do we want to specify the plot range to be the same for each graphic?

```

xLst = rand(1, 10);
yLst = rand(1, 10);
plot(xLst, yLst, 'ob', 'MarkerFaceColor', 'b')
axis([0 1 0 1])

```

Quick Review Question 3 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, "Constrained Growth"). We use the operators $.$ / $.$ and $.$ * so that the function works with arrays of numbers, such as $t = 0:0.1:3$, as well as numbers. To see the effect of increasing the rate of change from $(1 + 0.2(\mathbf{1})) = 1.2$ to $(1 + 0.2(\mathbf{10})) = 3.0$, replace the assignment of 5 to i with the start of a *for* loop to generate 10 plots of $f(t, i)$, where i varies from 1 to 10. Create a movie of the animation that plays five times.

```

clear('i', 'f', 't');
f = @(t, i) (1000*20)./((1000 - 20).*exp(-(1+0.2*i).*t) + 20);
t = 0:0.1:3;
i = 5; % replace with for loop
plot(t, f(t, i));
axis([0 3 0 1000]);

```

Function M-Files

Previously, we have used functions with very short definitions and, consequently, have defined them as anonymous functions. For example, we can define a function handle, sq , and an anonymous function for element-by-element squaring of a numeric or array parameter, as follows:

```
sq = @(x) x.*x;
```

However, for a function that has a longer definition or that we wish to reuse, we place the definition in a **function M-file**. To begin we select *New* and then *Function* from the *File* menu. The resulting file has a template for a function definition, such as follows:

```
function [ output_args ] = untitled( input_args )
%UNTITLED Summary of this function goes here
% Detailed explanation goes here

end
```

We can type the definition of a function to return the square of parameter as follows:

```
function [s] = sqr(x)
% SQR Square of vector elements
%   sqr(x) returns the element-by-element square of x.
s = x .* x;
end
```

The first line must contain the keyword *function*. Afterwards, the brackets surrounding *output_args* in the template indicate a vector or list of **output parameter(s)** or **output argument(s)**. In our case, the function should return one value, so we replace [*output_args*] with brackets and an output parameter, such as [s]. Alternatively, we can omit the brackets, as follows:

```
function s = sqr(x)
```

If a function does not return a value, we omit *output_args*. For our example, we follow [s] with an equals sign, the name of the function (here, *sqr*), and parentheses containing the **input parameter(s)** or **input argument(s)** (*input_args*, here, *x*). The next line, called an **H1 line**, contains the function name in all capitals, here *SQR*, and a comment describing what the function does. Then, **help text**, which begins with a typical function call, such as *sqr(x)*, describes how to use the function. If the user requests help concerning the function, as follows, *MATLAB* displays the H1 and help text lines:

```
help sqr
```

The resulting output is as follows:

```
SQR Square of vector elements
sqr(x) returns the element-by-element square of x.
```

Several lines can appear in the definition. However, only one line is contained in this definition in which the output parameter is assigned the element-by-element product of the input parameter with itself. We must be careful to always assign a value to the output variable.

If we have an error in the definition, a red line appears on the right side of the function window at the location where *MATLAB* detects a problem. Hovering the mouse over such a red line, *MATLAB* displays the error message. Keep in mind that the error may occur shortly before, often on the line above, the line on which the displayed error occurs. Yellow lines indicate warnings that may indicate a problem with the file.

After writing this function M-file, we save the file using the name of the function, here *sqr*, and *MATLAB* appends the extension *.m*. Thus, the file name is *sqr.m*. Once we save the function M-file, from the command window we can call the function with an argument of 4, as follows:

```
sqr(4)
```

We should get the answer of 16. However, we may get a message, such as follows:

```
??? Undefined function or method 'sqr' for input arguments
of type 'double'.
```

In this case, we need to inform *MATLAB* where to look for the definition. In the *File* menu, we choose *Set Path...*, click *Add Folder...*, browse to the appropriate folder, click *Open*, and *Save*. When we call a function, *MATLAB* searches the saved path names until finding a match. Because the software uses the first function by that name that it finds, we may need to adjust the search order of path names.

Quick Review Question 4

- Define a function, *rectCircumference*, that returns the circumference, *circumference*, of a rectangle with parameters for length and width, *l* and *w*, respectively. The circumference is $2l + 2w$. Use a function M-file and appropriate H1 line and help text.
- Remove the assignment to *circumference*, and if necessary, save the file to see errors and warnings. Hover your mouse over the red and/or yellow line(s) to the right, and record the error message(s). Correct the function definition and save.
- If necessary, set the path to the function definition.
- Call the function to return the circumference of a rectangle with dimensions 3 and 4.2, respectively.

Quick Review Question 5

- In a function M-file, define *randIntRange* with parameters *lower* and *upper* to return a random integer between *lower* and *upper* - 1, inclusively. Thus, *randIntRange*(-3, 3) should return a number from the set $\{-3, -2, -1, 0, 1, 2\}$.
- Write a *for* loop to display 10 random integers between 5 and 8, inclusively.

If we have several values to return, we can place them in a vector. For example, the following line begins a function definition in which the function returns the area and circumference of a circle with radius *r*:

```
function [area circumference] = circleStats(r)
```

In the function body, we have two lines for the area and circumference, as follows:

```
area = pi .* r .* r;
circumference = 2 * pi .* r;
```

We employ the operator *.** so that *circleStats* can operate on a vector of radii, such as *circleStats*([1 3]).

In calling the function, we assign the function call to a vector with appropriate variables, such as follows:

```
[ar cir] = circleStats(5)
```

Execution of the command assigns the area of the circle with radius 5 to *ar* and the circumference to *cir*.

Quick Review Question 6

- Define a function *squareStats* that returns the area (*side* squared) and circumference (four times *side*) of a square with a parameter, *side*, for the length of a side. Use a function M-file.
- Call the function to obtain the area and circumference of a square with each side having length 3.

By default, variables inside the definition are **local** to the function. To make symbols known elsewhere, we declare them to be **global**. However, we should use this feature with great care, because global variables can cause unexpected results, called **side effects**. It is appropriate to declare constants, usually written in all uppercase letters, that several functions use as global. If needed, the command window should also declare these constants as *global*. The following declares *EMPTY* to be a global variable and assigns it the value 0.

```
global EMPTY
EMPTY = 0;
```

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. A **logical OR operator** in *MATLAB* is `||`, so the statement is as follows:

```
if (x < -3) || (x > 3)
    disp('Out of bounds')
end
```

Although obviously not in this example, sometimes both conditions can be true, such as with $(x > 3) \parallel (y > 3)$ when *x* is 5 and *y* is 10. In this case, the compound condition is also true.

The logical OR operator employs **short circuiting**. With short circuiting, as soon as *MATLAB* can detect that a compound condition is true or false, the system stops computation of the condition. In the above example, $(x < -3) \parallel (x > 3)$, if the value of *x* is less than -3, we know that the compound condition is true; *MATLAB* does not need to evaluate the second condition, $x > 3$.

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

```
if (-3 <= x) && (x <= 3)
```

```

        disp('In bounds')
    end

```

We cannot write the condition as in mathematics, $(-3 \leq x \leq 3)$, but must express the condition with a compound statement. The operator `&&` also employs short circuiting. For example, if x is -5 , we know that the condition $(-3 \leq x)$ is false; and, in fact, we know the compound condition $(-3 \leq x) \&\& (x \leq 3)$ is false without calculating the second condition. Thus, because of short circuiting, *MATLAB* would not evaluate $(x \leq 3)$.

To negate a condition, we employ the **logical NOT operator** `~`. Thus, $\sim(x > 3)$ is equivalent to $(x \leq 3)$.

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 *MATLAB* evaluates an inequality before evaluating an OR operator:

```

    if x < -3 || x > 3
        disp('Out of bounds')
    end

```

Table 1 Operator precedence from highest to lowest

1.	()					
2.	'	^	!	.	^	
3.	Unary +	Unary -	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 *ismember* determines if an expression is a member of an array or not. A general form is as follows:

```
ismember(expr, array)
```

The command returns *true* (1) if *expr* is a member of *array* and *false* (0) 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 vector *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
    statements
end
```

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 = rand
while ra < 0.7
    counter = counter + 1;
    ra = rand
end
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 1 to *i*, an index. While *x* is between -5 and 5, plot the point (*x*, 0) as a large red dot; save the frame as the *i*th element of a vector; add 1 to *i*; and use *randi* or *randIntRange* from Quick Review Question 5 to generate a random integer -1, 0, or 1 and assign to *x* the sum of this number and *x*. After the loop, show the animation again.