A BRIEF INTRODUCTION TO MATLAB

Fotios Kasolis
2008-2009

Contents

1. Introduction

+ | - | * | / | ^ | pi | i | j | format | sin | cos | tan | sinh | cosh | tanh | asin | acos | atan | asinh | acosh | atanh | exp | log | abs | sqrt | sign

We can use MATLAB to do arithmetic as we would with a common calculator, to do so we use + to add, - to subtract, * to multiply, / to divide and ^ to exponentiate.

```
>>3^2+(2*4-1)/7↲
ans =
        10
```

MATLAB prints the answer and assigns the value to the variable ans. We can use the variable ans to perform further calculations.

```
>>ans^2↲
ans =
        100
```

MATLAB assigns a new value to the variable ans in each calculation. In the example above if we ask MATLAB the value of ans by typing:

```
>>ans↲
ans =
        100
```

we see that ans has the value of our last calculation.

The format of the displayed output can be controlled by the format command with basic arguments short (default), long and rat.

```
>>format long↲
>>pi↲
ans =
        3.14159265358979
```

```
>>format rat↲
>>pi↲
ans =
        355/113
>>format short↲
>>
```

2. Asking for help

help | lookfor

There are several ways to get help in MATLAB. To get help on a particular command, enter help followed by the name of the command. For example,

```
>>help solve↲
```

will display documentation for the solve command. The command lookfor searches the first line of every MATLAB help file for a specified string (use lookfor -all to search all lines). For example, if you wanted to see a list of all MATLAB commands that contain the word fft as part of the command name or brief description, then you would type:

```
>>lookfor fft↲
```

If the command you are looking for appears in the list, then you can use help on that command to learn more about it.

3. Variables and assignments

= | ; | clear | whos

In MATLAB, we use the equal sign = to assign values to a variable. For instance,

```
>>x=1↲
x =
        1
```

while we can force MATLAB not to print the result of the assignment by using ; like in the example below.

```
>>y=2;↲
>>
```

In both cases the value in the rhs is assigned to the variable in the lhs, that is variable = value. Now we can use the assigned variables in order to do more complex calculations.

```
>>x^2+y^2↲
ans =
        5
```

To clear the value of the variable x we type:

```
>>clear x↲
>>
```

A common source of puzzling errors is the inadvertent reuse of previously defined variables. MATLAB never forgets your definitions unless instructed to do so. You can check on the current value of a variable by simply typing its name.

We can type whos to see a summary of the names and types of the currently defined variables.

```
>>whos↵
Name     Size              Bytes  Class

 x       1x1                   8  double array
 y       1x1                   8  double array

Grand total is 2 elements using 16 bytes
```

A variable name or function name can be any string of letters, digits, and underscores, provided it begins with a letter (punctuation marks are not allowed). MATLAB distinguishes between uppercase and lowercase letters. You should choose distinctive names that are easy for you to remember, generally using lowercase letters.

4. Relational operators

< | > | <= | >= | == | ~= | & | | | ~ | && | ||

MATLABs' relational operators are given in the table below.

>	less than
<	greater than
<=	less than or equal
>=	greater than or equal
==	equal
~=	not equal

They all operate entry-wise. Note that = is used in an assignment statement whereas == is a relational operator. Relational operators may be connected by logical operators:

&	and
|	or
~	not
&&	short circuit and
||	short circuit or

The result of a relational operator is of type logical, and is either true (one) or false (zero). Thus, ~0 is 1, ~3 is 0, and 4 & 5 is 1, for example. When applied to scalars, the result is a scalar. Try entering 3 < 4, 3 > 4, 3 == 4, and 3 == 3. When applied to matrices of the same size, the result is a matrix of ones and zeros giving the value of the expression between corresponding entries.

5. Vectors and matrices

: | ’ | .* | ./ | .^ | zeros | ones | eye | \ | eig | diag | triu | tril | rand | hilb | magic | poly | det | size | length | norm | rank

●**Vectors.** A vector is an ordered list of numbers. You can enter a vector of any length in MATLAB by typing a list of numbers, separated by commas or spaces, inside square brackets.

```
>>A=[1 2 3 4 5 6]↲
A =
     1   2   3   4   5   6
>>B=[1,2,3,4,5,6]↲
B =
     1   2   3   4   5   6
```

Suppose you want to create a vector of values running from 0 to 5. Here’s how to do it without typing each number.

```
>>C=0:5↲
C =
     0   1   2   3   4   5
```

The increment can be specified as the second of three arguments.

```
>>D=0 : .2 : 1↲
D =
     0   0.2000   0.4000   0.6000   0.8000   1.0000
```

To change the vector C from a row vector to a column vector, put a prime (’) after C.

```
>>c=C’↲
c =
     0
     1
     2
     3
     4
     5
```

We can perform mathematical operations on vectors. For example, to square the elements of the vector C, type:

```
>> C .^ 2↲
ans =
     0   1   4   9   16   25
```

The period in this expression is very important; it says that the numbers in C *should be squared individually, or element by element. Typing* C^2 *would tell MATLAB to use matrix multiplication to multiply* C *by itself and would produce an error message in this case. Similarly, we must type .* or ./ if we want to multiply or divide vectors element by element.*

Most MATLAB operations are, by default, performed element by element. For example, we do not type a period for addition and subtraction, and you can type exp(C) to get the exponential of each number in C (the matrix exponential function is expm).

●**Matrices.** A matrix is a rectangular array of numbers. Row and column vectors, which we discussed above, are examples of matrices. Consider the 2×2 matrix:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

It can be entered in MATLAB with the command:

```
>>A=[1 2 ; 3 4]↲
A =
     1   2
     3   4
```

with elements:

```
>>A(1,1)↲
ans =
     1
>>A(1,2)↲
ans =
     2
>>A(2,1)↲
ans =
     3
>>A(2,2)↲
ans =
     4
```

Moreover we can extract rows and/or columns by using colon notation like in the examples below.

```
>>A(1,:)↲
ans =
     1   2
>>A(:,2)↲
ans =
     2
     4
```

A colon by itself denotes an entire row or column.

We can redefine an element in a predefined matrix A by typing:

```
>>A(1,1)=5↲
A =
     5   2
     3   4
```

while in the same way we can redefine a hole row and/or column.

```
>>A(1,:)=[3 3]↲
A =
     3   3
     3   4
```

MATLAB has many commands for manipulating matrices. You can read about them in the online help; some of them are illustrated in the text.

In addition to the usual algebraic methods of combining matrices (e.g., matrix multiplication), we can also combine them element-wise. Specifically, if A and B are of the same size, then A.*B is the element-by-element product of A and B, that is, the matrix whose (i,j) element is the product of the (i,j) elements of A and B. Likewise, A./B is the element by element quotient of A and B, and A.^c is the matrix formed by raising each of the elements of A to the power c. More generally, if f is one of the built-in functions in MATLAB, or is a user-defined function that accepts vector arguments, then f(A) is the matrix obtained by applying f element by element to A. See what happens when you type sqrt(A), where A is the matrix defined at the beginning of the Matrices section. Recall that V(3) is the third element of a vector V. Likewise, A(2,3) represents the (2,3) element of A, that is, the element in the second row and third column. You can specify sub matrices in a similar way. Typing A(2,[2 4]) yields the second and fourth elements of the second row of A. To select the second, third, and fourth elements of this row, type A(2,2:4). The sub matrix consisting of the elements in rows 2 and 3 and in columns 2, 3, and 4 is generated by A(2:3,2:4). The commands zeros(n,m) and ones(n,m) produce n×m matrices of zeros and ones, respectively. Also, eye(n) represents the n×n identity matrix.

Suppose A is a nonsingular n×n matrix and b is a column vector of length n. Then typing x = A\b numerically computes the unique solution to A*x = b. Type help mldivide for more information. If either A or b is symbolic rather than numeric, then x = A\b computes the solution to A*x = b symbolically. To calculate a symbolic solution when both inputs are numeric, type x = sym(A)\b.

The eigenvalues of a square matrix A are calculated with eig(A). The command [V, r] = eig(A) calculates both the eigenvalues and eigenvectors. The eigenvalues are the diagonal elements of the diagonal matrix r, and the columns of V are the eigenvectors.

6. Control flow statements

for | while | if | switch | try/catch

In their basic form, these MATLAB flow control statements operate like those in most computer languages. Indenting the statements of a loop or conditional statement is optional, but it helps readability to follow a standard convention.

● **for.** In the example below the vector x grows in size at each iteration 1:10. Note that a matrix may be empty, such as x = [].

```
%example1.m
x=[];
for i=1:10
        x=[x,i^2];
end
x
>>example1↲
x =
   1   4   9   16   25   36   49   64   81   100
```

- **while.** The “statements” will be repeatedly executed as long as the “expression” remains true.

```
while “expression”
        “statements”
end
```

- **if.** The “statements” will be executed only if the “expression” is true.

```
if “expression-1”
        “statement-1”
elseif “expression-2”
        “statement-2”
…
else
        “statement-n”
end
```

- **switch.** The switch statement is just like the if statement. If you have one expression that you want to compare against several others, then a switch statement can be more concise than the corresponding if statement. See help switch for more information.
- **try/catch.** Matrix computations can fail because of characteristics of the matrices that are hard to determine before doing the computation. If the failure is severe, your script or function may be terminated. The try/catch statement allows you to compute optimistically and then recover if those computations fail. The general form is:

```
try
        “statements-1”
catch
        “statements-2”
end
```

The first block of statements is executed. If an error occurs, those statements are terminated, and the second block of statements is executed. You cannot do this with an if statement.

7. M-files

function

MATLAB can execute a sequence of statements stored in files. These files are called M-files because they have the extension .m, moreover we distinguish them in:

- **scripts**: consists of a sequence of normal MATLAB statements.
- **functions**: Function files provide extensibility to MATLAB. You can create new functions specific to your problem, which will then have the same status as other MATLAB functions. Variables in a function file are by default local.

Typing abcd in the command window causes the statements in the script file abcd.m to be executed.

```
%exampl2.m
a=[1 2 ; 3 4];
b=[5 6 ; 7 8];
a*b
```

```
>>example2↲
ans =
        19   22
        43   50
```

You often need to repeat a process several times for different input values of a parameter. For example, you can provide different inputs to a built-in function to find an output that meets a given criterion. Let us describe a problem, where we want to compute some values of sin(x)/x with x = 10−b for several values of b. Suppose, in addition, that you want to find the smallest value of b for which sin(10−b)/(10−b) and 1 agree to 15 digits.

```
%sinelimit.m
%sinelimit computes sin(x)/x for x = 10^(-b),
%where b = 1,...,c.
function y = sinelimit(c)
format long
b = 1:c;
x = 10.^(-b);
y = (sin(x)./x)';
>>sinelimit(10)↲
ans =
        0.99833416646828
        0.99998333341667
        0.99999983333334
        0.99999999833333
        0.99999999998333
        0.99999999999983
        1.00000000000000
        1.00000000000000
        1.00000000000000
        1.00000000000000
>>help sinelimit↲
sinelimit.m
sinelimit computes sin(x)/x for x = 10^(-b),
where b = 1,...,c.
```

Like a script M-file, a function M-file is a plain text file that should reside in your MATLAB working directory. The first line of the file contains a function statement, which identifies the file as a function M-file. The first line specifies the name of the function and describes both its input arguments (or parameters) and its output values. In this example, the function is called sinelimit. The file name and the function name should match. The function sinelimit takes one input argument and returns one output value, called c and y (respectively) inside the M-file. When the function finishes executing, its output will be assigned to the trivial variable ans (by default) or to any other variable you choose, just as with a built-in function. The remaining lines of the M-file define the function. In this example, b is a row vector consisting of the integers from 1 to c. The vector y contains the results of computing sin(x)/x where x = 10−b; the prime makes y a column vector. Notice that the output of the lines defining b, x, and y is suppressed with a semicolon. In general, the output of intermediate calculations in a function M-file should be suppressed.

Of course, when we run the M-file above, we do want to see the results of the last line of the file, so a natural impulse would be to avoid putting a semicolon on this last line. But because this is a function M-file, running it will automatically display the contents of the designated output variable y. Thus if we did not put a semicolon at the end of the last line, we would see the same numbers twice when we run the function!

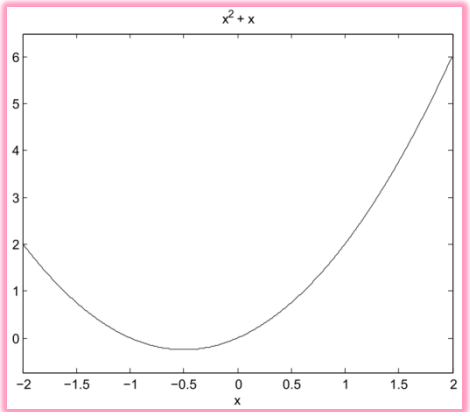
Note that the variables used in a function M-file, such as b, x, and y in sinelimit.m, are local variables. This means that, unlike the variables that are defined in a script M-file, these variables are completely unrelated to any variables with the same names that you may have used in the command window, and MATLAB does not remember their values after the function M-file is executed.

8. 2D and 3D graphs

ezplot | plot | plot3 | subplot | quiver | polar | hold | surf | mesh | contour

The simplest way to graph a function of one variable is with ezplot, which expects a string or a symbolic expression representing the function to be plotted. For example, to graph x^2+x on the interval −2 to 2, type:

```
>>ezplot('x^2 + x',[-2 2])↲
```



The plot will appear on the screen in a new labeled window. We mentioned that ezplot accepts either a string argument or a symbolic expression. Using a symbolic expression, you can produce the next plot with the following input.

```
>> syms x↲
>> ezplot(x^2 + x, [-2 2])↲
```

Graphs can be misleading if you do not pay attention to the axes. For example, the input ezplot(x^2+x+3, [-2 2]) *produces a graph that looks identical to the previous one, except that the vertical axis has different tick marks (and MATLAB assigns the graph a different title).*

You can modify a graph in a number of ways. You can change the title above the graph in Figure 2-4 by typing in the command window:

```
>>title 'A Parabola'↲
```

You can add a label on the horizontal axis with xlabel or change the label on the vertical axis with ylabel. Also, you can change the horizontal and vertical ranges

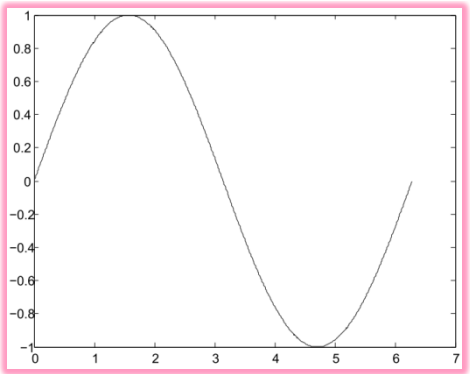
of the graph with axis. For example, to confine the vertical range to the interval from 1 to 4, type:

```
>>axis([-2 2 1 4])↲
```

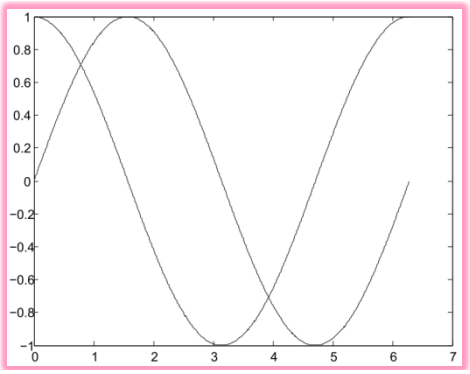
The first two numbers are the range of the horizontal axis; both ranges must be included, even if only one is changed.

The command plot works on vectors of numerical data. The basic syntax is plot(x,y) where x and y are vectors of the same length.

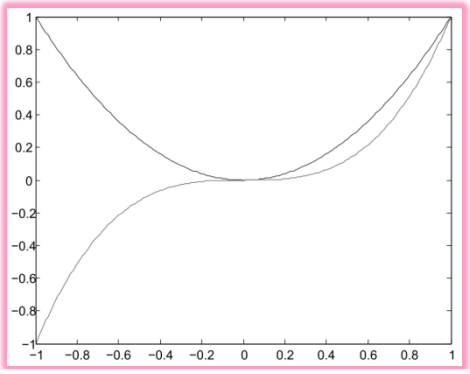
```
>>x=0:.01:2*pi;↲
>>y=sin(x);↲
>>plot(x,y)↲
```



```
>>hold on↲
>>plot(x,cos(x))↲
```

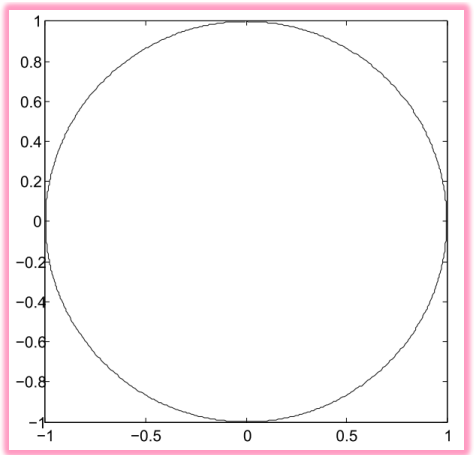


```
>>x=-1:.01:1;↲
>>plot(x,x.^2,x,x.^3)↲
```



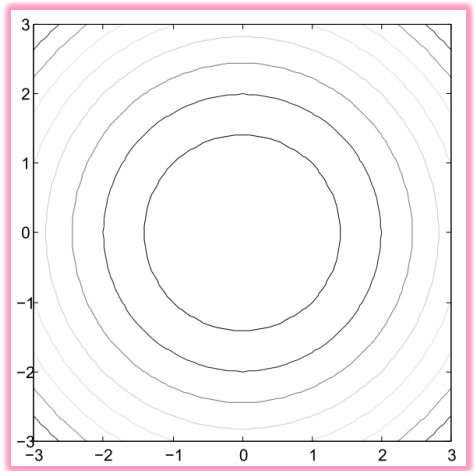
- **Parametric plot**

```
>>t=0:.01:2*pi;↲
>>plot(sin(t),cos(t))↲
>>axis square↲
```



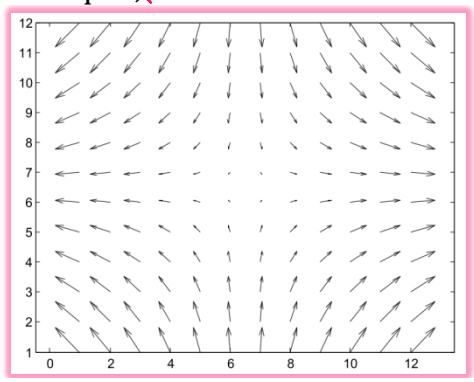
- **Contour plot**

```
>>[x y] = meshgrid(-3:0.1:3, -3:0.1:3);↲
>>contour(x, y, x.^2 + y.^2)↲
>>axis square↲
```



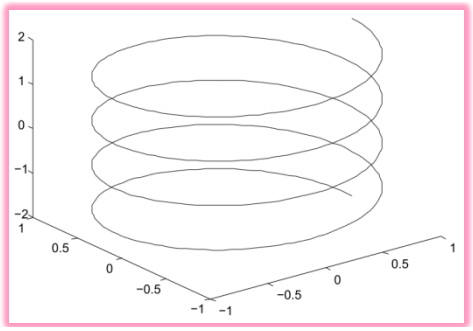
- **Field plot**

```
>>[x, y] = meshgrid(-1.1:.2:1.1, -1.1:.2:1.1);↲
>>quiver(x, -y); axis equal;↲
```



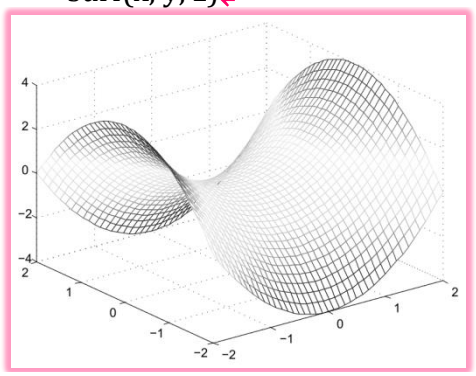
For plotting curves in 3-space, the basic command is plot3, and it works like plot, except that it takes three vectors instead of two, one for the x coordinates, one for the y coordinates, and one for the z coordinates.

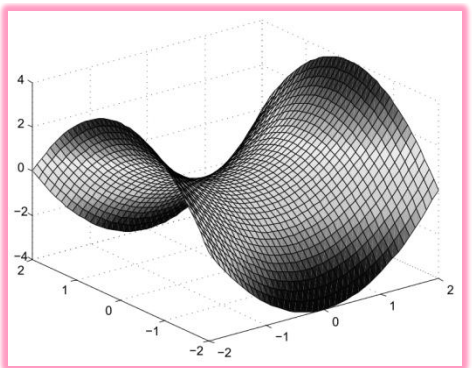
```
>>t = -2:0.01:2;↲
>>plot3(cos(2*pi*t), sin(2*pi*t), t)↲
```



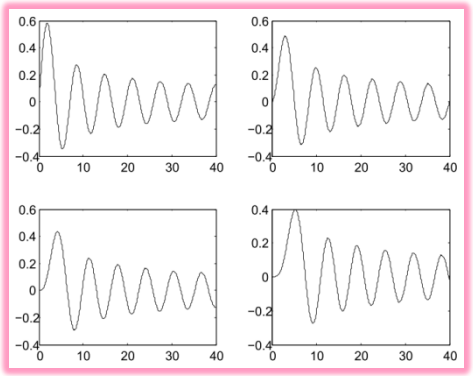
There are two basic commands for plotting surfaces in 3-space: mesh and surf. The former produces a transparent "mesh" surface; the latter produces an opaque shaded one. There are two different ways of using each command, one for plotting surfaces in which the z coordinate is given as a function of x and y, and one for parametric surfaces in which x, y, and z are all given as functions of two other parameters. To plot z = f (x, y), one begins with a meshgrid command as in the case of contour. For example, the saddle surface $z = x^2 - y^2$ can be plotted with:

```
>>[x, y] = meshgrid(-2:.1:2, -2:.1:2);↲
>>z = x.^2-y.^2;↲
>>mesh(x, y, z)↲
>>surf(x, y, z)↲
```





The command subplot divides the figure window into an array of smaller figures. The first two arguments give the dimensions of the array of subplots, and the last argument gives the number of the subplot (counting left to right across the first row, then left to right across the next row, and so on) in which to put the next figure.



```
>>x = 0:0.05:40;↲
>>for j = 1:4, subplot(2,2,j)↲
plot(x, besselj(j*ones(size(x)), x))↲
end↲
```

9. Programming

disp | input | error | nargin | nargout | return | eval | feval

●In general, loops should be avoided in MATLAB, as they can significantly increase the execution time of a program. MATLAB has the capability to increase the size of vectors and matrices dynamically, as can be required by a for loop. For example, a vector can be of length N at the Nth iteration of a loop, and at iteration N+1, MATLAB can increase the length of the vector from N to N+1. However, this dynamic storage allocation is not accomplished efficiently in terms of computation time. For example, consider the following script for generating a sine wave:

```
% sinusoid of length 5000
for t=1:5000
        y(t)=sin(2*pi*t/10);
end
```

This results in a scalar t, with final value 5000, and a vector y, a sine wave with 10 samples per cycle, with 5000 elements. Each time the command inside the for loop is executed, the size of the variable y must be increased by one. The execution time on a certain PC is 7.91 seconds. Then consider a second script:

```
% sinusoid of length 10000
for t=1:10000
        y(t)=sin(2*pi*t/10);
end
```

Clearing the MATLAB workspace and running this script to produce vector y having 10000 elements requires an execution time on the same PC of 28.56 seconds, nearly four times the execution time of the first script. Forcing MATLAB to allocate memory for y each time through the loop takes time, causing the execution time to grow geometrically with the vector length.

To maximize speed, arrays should be pre-allocated before a for loop is executed.

To eliminate the need to increase the length of y each time through the loop, the script could be rewritten as:

```
% sinusoid of length 10000 with vector pre-allocation
y=zeros(1,10000);
for t=1:10000
        y=sin(2*pi*t/10);
end
```

In this case, y was arbitrarily defined to be a vector of length 10000, with values recomputed in the loop. The computation time in this case was 2.03 seconds. Execution time can be decreased even further by “vectorizing” the algorithm, applying an operation to an entire vector instead of a single element at a time. Applying this approach to our example of sinusoid generation:

```
% better method of sinusoid generation
t=1:10000;
y=sin(2*pi*t/10);
```

This script produces the vector t with 10000 elements and then the vector y with 10000 elements, which is the same y from the second script above. However,

again using the same PC, this script executes in 0.06 second, as opposed to 16.04 seconds for the second for loop script, or 1.92 seconds with a for loop and vector pre-allocation.

- The while loop allows the execution of a set of commands an indefinite number of times. The break command is used to terminate the execution of the loop. We use a while loop to evaluate quadratic polynomials. Consider writing a script to ask the user to input the scalar values for a, b, c, and x and then returns the value of ax^2+bx+c. The program repeats this process until the user enters zero values for all four variables.

```
% Script to compute ax^2+bx+c
disp('Quadratic ax^2+bx+c evaluated')
disp('for user input a, b, c, and x')
a=1; b=1; c=1; x=0;
while a~=0 | b~=0 | c~=0 | x~=0
        disp('Enter a=b=c=x=0 to terminate')
        a=input('Enter value of a:');
        b=input('Enter value of b:');
        c=input('Enter value of c:');
        x=input('Enter value of x:');
        if a==0 & b==0 & c==0 & x==0
                break
        end
        quadratic=a*x^2+b*x+c;
        disp('Quadratic result:')
        disp(quadratic)
end
```

Note that this script will display multiple quadratic expression values if arrays are entered for a, b, c, or x. Consider modifying the script to check each input and breaking if any input is not a scalar. The function error displays a character string in the command window, aborts function execution, and returns control to the keyboard. This function is useful for flagging improper command usage, as in the following portion of a function:

```
if length(val)>1
        error('val must be a scalar.')
end
```

- The functions nargin and nargout can be used in functions to determine the number of input arguments and the number of output arguments used to call the function. This provides information for handling cases when the function is to use a variable number of arguments. For example, if a default value of 1 is to be returned for the output variable err if the function call includes no input arguments, the function can include the following:

```
if nargin==0
        err=1;
end
```

If the function is written to use a fixed number of input or output arguments, but the wrong number has been supplied in the use of the function, MATLAB provides the needed error handling, without the need for the nargin and nargout functions.

●Function M-files terminate execution and return when they reach the end of the M-file, or alternatively, when the command return is encountered.
●The commands eval and feval allow you to run a command that is stored in a string as if you had typed the string on the command line. If the entire command you want to run is contained in a string str, then you can execute it with eval(str). For example, typing eval('cos(1)') will produce the same result as typing cos(1). Generally eval is used in an M-file that uses variables to form a string containing a command; see the online help for examples. You can use feval on a function handle or on a string containing the name of a function you want to execute. For example, typing feval('atan2',1,0) or feval(@atan2,1,0) is equivalent to typing atan2(1,0). Often feval is used to allow the user of an M-file to input the name of a function to use in a computation. The following M-file iterate.m takes the name of a function and an initial value and iterates the function a specified number of times:

```
function final=iterate(func, init, num)
final=init;
for k=1:num
        final=feval(func, final);
end
```

Typing iterate('cos',1,2) yields the numerical value of cos(cos(1)), while iterate('cos', 1, 100) yields an approximation to the real number x for which cos(x)=x (Think about it!). Most MATLAB commands that take a function name argument use feval, and as with all these commands, if you give the name of an inline function to feval, you should not enclose it in quotes.

10.Spinning coins

rand

When a fair (unbiased) coin is spun, the probability of getting heads or tails is 0.5 (50%). Since a value returned by rand is equally likely to anywhere in the interval [0,1) we can represent heads, say, with a value less than 0.5, and tails otherwise. Suppose an experiment calls for a coin to be spun 50 times, and the results recorded. In real life you may need to repeat such an experiment a number of times; this is where computer simulation is handy. The following script simulates spinning a coin 50 times.

```
for i = 1:50
        r = rand;
        if r < 0.5
                fprintf( 'H' )
        else
                fprintf( 'T' )
        end
end
fprintf( '\n' )      % newline
```

Here is the output from two sample runs:

```
THHTTHHHHTTTTTHTHTTTHHTHTTTHHTTTTHTTHHTHHHHHHTTHTT
THTHHHTHTHHTTHTHTTTHHTTTTTTTHHHTTTHTHHTHHHHTTHTHTT
```

Note that it should be impossible in principle to tell from the output alone whether the experiment was simulated or real (if the random number generator is sufficiently random). Can you see why it would be wrong to code the if part of the coin simulation like this:

if rand < 0.5 fprintf('H'), end
if rand >= 0.5 fprintf('T'), end

The basic principle is that rand should be called only once for each event being simulated. Here the single event is spinning a coin, but rand is called twice. Also, since two different random numbers are generated, it is quite possible that both logical expressions will be true, in which case H and T will both be displayed for the same coin!

References

1. MATLAB primer 7th. Timothy A. Davis, Kermit Sigmon. CRC.
2. A guide to MATLAB. Brian R. Hunt, Ronald L. Lipsman, Jonathan M. Rosenberg. Cambridge university press.
3. Essential MATLAB. Brian D. Hahn, Daniel T. Valentine. BH Elsevier.
4. Advanced mathematics and mechanics using MATLAB. Howard B. Wilson, Louis H. Turcotte, David Halpern. CHAPMAN & HALL/CRC