An Introduction to MATLAB

Jeffery Cooper
Department of Mathematics
University of Maryland

Contents

Chapter 1 Basic MATLAB

1.1 First steps
1.2 Vectors and Matrices
1.3 Array operations
1.4 Matrix multiplication and linear systems
1.5 MATLAB functions and mfiles
1.6 Two dimensional graphs

Chapter 2 Basic MATLAB Continued

2.1 Some very useful commands
2.2 Script mfiles and programs
2.3 Operators on functions
2.4 Functions of two variables
2.5 MATLAB documents

Chapter 3 More Features of MATLAB

3.1 The command `feval`
3.2 Vectorizing computations
3.3 Progamming logic

1 Basic MATLAB

1.1 First Steps

When you invoke MATLAB to begin a session, you see the prompt

```
>>
```

When we give instructions for an operation, or request information, following this prompt, we say that we are working on the *command line*

We can do simple arithmetic operations on the command line such as $(2 + 3.5^2 - 4.7)/12$,

```
>> (2+3.5^2 -4*7)/12
ans =
  -1.1458
```

We can also do this calculation by assigning variable names to the quantities.

```
>> x = 2+3.5^2
ans =
  14.2500
>> y = 4*7
ans =
  28
>> z = (x-y)/12
ans =
  -1.1458
```

If we do not wish to see the intermediate results, we can suppress the numerical output by putting a semicolon at the end of the line. Then the sequence of commands and output looks like this.

```
>> x = 2+3.5^2;
>> y = 4*7;
1>> z = (x-y)/12;
>> z
z =
   -1.1458
```

MATLAB does its numerical calculations in double precision, which is 15 digits. Normally only five digits are displayed. If we want to see all 15 digits, we use the command `format long`.

```
>> format long
>> z
z =
  -1.14583333333333
```

To return to the short format, enter `format short`.

Exiting

To leave MATLAB enter `quit`.

If MATLAB gets hung up in calculation, or is taking a long time, and you want to stop the calculation, without exiting MATLAB, enter Ctrl+C.

1.2 Vectors and matrices

Forming vectors and matrices

Matrices can be entered by typing in the elements one at a time. Typing this

```
>> [1 2 3;4 5 6]
```

produces

```
ans =
    1     2     3
    4     5     6
```

Notice that we use a semicolon to separate the rows. Usually we want to give a vector or matrix a name. To assign a matrix value to a variable we proceed as follows: Type this

```
>> A = [1 2 3;3 4 5]

A =
   1     2     3
   4     5     6
```

Remember, to suppress the output, put a semicolon after the defining statement. This can be especially important if the matrix or vector has thousands of elements.

```
>> A = [1 2 3;4 5 6];
```

If we want to see the numbers in `A` we type

```
>> A
```

which produces

```
A =
   1     2     3
   4     5     6
```

The transpose of a real matrix is formed by the command `A'`. If the row vector `x` is defined by

```
>> x = [1 5 4 8 10]
```

then `x` is turned into a column vector with the command `x'`. If the matrix or vector has complex elements, $\mathbf{A}'$ is the Hermitean transpose, which is the transpose with the complex conjugate of the elements. For example

```
Z =
    1+i   2    1
    2+5i  i    2
>> Z'
Z'=
    1-i  2-5i
     2    -i
     1    2
```

To get a transpose, without taking the complex conjugates, use a dot before the apostrophe: `A.'`.

To determine the dimensions of a vector or matrix, use the commands

```
>> size(A)
ans =
     2     3

>> size(x)
ans =
     1     5

>> size(x')
     5     1
```

To view a certain element in a matrix or vector we specify its location with a command

```
>> A(1,2)
ans =
     2

>> x(5)
ans =
     10
```

In many cases the vectors or matrices are far too large to enter one element at a time. For instance if we want to enter a vector **x** consisting of points $(0.1, .2, .3, .4, \ldots, 5.5, 6)$ we can use the command

```
>> x = 0:.1:6 ;
```

This row vector has 61 elements. Another way to create the same vector is to use the command `linspace`.

```
>> x = linspace(0,6,61);
```

`linspace` stands for "linear spacing". It is useful when we want to divide an interval into a number of subintervals of the same length. For example, `theta = linspace(0, 2*pi, 41)` divides the interval $[0, 2\pi]$ into 40 equal subintervals, creating a vector of 41 elements.

To create a vector of zeros or of ones of the same length as a give vector x, there are commands

```
>> y = ones(size(x));
>> z = zeros(size(x));
```

The same works for matrices

```
>> Z = zeros(size(A));
>> Y = ones(size(A))
Y =
     1    1    1
     1    1    1
```

One can also specify a matrix of zeros or ones by giving the dimensions.

```
>> Z = zeros(2,3)
```

The $n \times n$ identity matrix is produced with the command `eye(n)`. There are special commands for entering sparse matrices or diagonal matrices. For more information, enter `help sparse` or `help diag`.

1.3 Operations on matrices

Arithmetic of matrices

Addition and subtraction of matrices are done in the obvious way.

```
>> B = [2 0 -1; 1 2 7];
>> A + B
ans =
     3     2     2
     5     7    13
```

In general MATLAB can add together only matrices having the same dimension. However, there is one special and very useful exception. If $\mathbf{A}$ is a matrix and c is a scalar, then the sum `A + c` means to add c to every element of $\mathbf{A}$. In particular if x is a vector and t a scalar then `x+t` is a vector of the same length with t added to each component.

A vector or matrix can always be multiplied or divided by a scalar.

```
>> 2 * A
ans =
      2     4     6
      8    10    12

>> A/2
ans =
    0.5000    1.0000    1.5000
    2.0000    2.5000    3.0000
```

Array operations

Arithmetic operations can also be performed on matrices, entry by entry. These are called array operations. Array multiplication is an example. If $\mathbf{A}$ and $\mathbf{B}$ are two matrices of the same size with elements $a_{i,j}$ and $b_{i,j}$, then the symbol

```
>> C = A.*B
```

produces another matrix $\mathbf{C}$ of the same size with elements $c_{i,j} = a_{i,j}b_{i,j}$. For example using the same 2×3 matrices $\mathbf{A}$ and $\mathbf{B}$ we defined earlier, we have

```
>> C = A.*B
C =
    2     0    -3
    4    10    42
```

To raise a scalar to a power, say two, we use the command `5^2`. If we want the operation to be applied to each element of a matrix, we use `.^2`. For example, if we want to produce a new matrix whose elements are the square of the elements of the matrix **A** we enter

```
>> A.^2
ans =
     1     4     9
    16    25    36
```

There is also a kind of array division for two matrices of the same size which divides the two matrices element by element.

```
>> D = [1 3 5; -2 4 -1]
>> A./D
ans =
    1.0000    0.6667    0.6000
   -2.0000    1.2500   -6.0000
```

1.4 Matrix multiplication and linear systems

Another kind of multiplication between matrices is motivated by the consideration of linear systems of equations. Let **A** be a 3×3 matrix

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

and

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

a 3×1 column vector. We define the product **Ax** to be a 2×1 column vector with components

$$\begin{bmatrix} a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 \\ a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 \end{bmatrix}.$$

With this defintion of multiplication of a matrix by a vector, we can write the linear system of two equations in the three unknowns $x_1,\ x_2,\ x_3$

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 &= b_2 \end{aligned}$$

as simply

$$\mathbf{Ax} = \mathbf{b}$$

where $\mathbf{b}$ is the 2×1 column vector

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}.$$

More generally if $\mathbf{A} = [a_{i,j}]$ is an $m \times n$ matrix, and $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ is an $n \times 1$ column vector, we define $\mathbf{Ax}$ to be the $m \times 1$ column vector with ith component

$$\sum_{j=1}^{n} a_{i,j}x_j.$$

In this way, the system of m linear equations in n unknowns x_j,

$$\sum_{j=1}^{n} a_{i,j}x_j = b_i, \quad i = 1, \ldots, m$$

can be written compactly as

$$\mathbf{Ax} = \mathbf{b}. \tag{1.1}$$

Now let $\mathbf{A}$ be an $m \times n$ matrix and $\mathbf{B}$ be an $n \times p$ matrix. We label the columns of $\mathbf{B}$ as $\mathbf{B}_j = [b_{i,j}],\ i = 1, \ldots, p$. We define

$$\mathbf{AB} = \mathbf{C} \tag{1.2}$$

where $\mathbf{C}$ is the $m \times p$ matrix whose columns are the $m \times 1$ column vectors $\mathbf{C}_j = \mathbf{AB}_j$, $j = 1, \ldots, p$. In terms of the entries,

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k}b_{k,j}.$$

This matrix multiplication $\mathbf{AB}$ is only defined for an $m \times n$ matrix $\mathbf{A}$ and an $n \times p$ matrix $\mathbf{B}$. The column dimension of $\mathbf{A}$ must equal the row dimension of $\mathbf{B}$.

In MATLAB we can multiply matrices in this fashion with the symbol `*`. It is very important to notice that this kind of matrix operation uses the symbol `*`, without the dot in front. `.*` is the symbol for array multiplication. We assume we have matrices of the correct dimensions.

```
>> A = [1 2; 3 3; 4 5];
>> B = [-1 3; 5 1];
>> C = A*B;
>> C
=  9   5
  12  12
  21  17
```

If $\mathbf{A}$ is a square matrix, $n \times n$, $\mathbf{A}$ can be multiplied times itself any number of times. We use the notation $\mathbf{A}^k$ to denote the product of k factors $\mathbf{A}\mathbf{A}\ldots\mathbf{A}$. The MATLAB command for raising a matrix to a power is `A^k`. Notice that the command does not have the dot in front. `A.^k` means the array operation which raises each element of $\mathbf{A}$ to the k^{th} power.

Given an $n \times n$ matrix $\mathbf{A}$ and an n column vector $\mathbf{b}$, the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ can be solved in several ways. The simplest way is to use the following method.

```
>> A = [1 2 3; 4 5 6; 6 7 9];
>> b = [ 1 0 1]';
>> x = A\b;
x =
   -0.0000
   -2.0000
    1.6667
```

The key command is `A\b`. MATLAB uses the method of Gaussian elimination with partial pivoting to solve linear systems.

1.5 MATLAB functions and mfiles

MATLAB has the usual built in functions such as $\sin x$, $\cos x$, $\tan x$, $\exp x$, $\log x$, $\sqrt{x}$, etc. These functions can take matrices as arguments, in which case the function is applied to each element of the matrix. We say that such a function is *array-smart*. For example, the cosine function can be applied to a matrix:

```
>> T = [2 3 pi; 8 pi/2 1];
>> cos(T)
ans =
   -0.4161   -0.9900  -1.0000
   -0.1455    0.0000   0.5403

>> sqrt(A)
```

```
ans =
    1.0000    1.4142    1.7321
    2.0000    2.2361    2.4495
```

In addition many other specialized functions are available. These include the error function, called by `erf(x)`, and Bessel functions of all orders. There are also functions of linear algebra which find information about matrices, such as `eig(A)` which finds the eigenvalues of a matrix **A**.

Often, however, we will need to build our own functions of one, two, or three variables. In this section we shall only consider functions of one variable. Functions of several variables will be discussed in the next chapter.

We can build our own functions in two ways. The first way is done on the command line, and is called an *inline function*. Here is a simple example.

```
>> f = inline('x^3 +x -1')
```

To evaluate $f(x) = x^3 + x - 1$ at $x = 2$, enter `>> f(2)`. If we wish the function to be array-smart, we must write

```
>> f = inline('x.^3 +x -1')
```

Notice that we have replaced `x^2` by `x.^2`. Functions created this way can take vectors and matrices as arguments. The function will be applied to each element of the vector or matrix. For example if the matrix **A** is given by

```
A =
     1    2    3
     4    5    6
```

then,

```
>> B = f(A)

B =  1    9    29
    67  129   221
```

The second important way to define new functions in MATLAB uses the extremely flexible MATLAB feature known as an *mfile*. There are two kinds of mfiles, *function* mfiles and *script* mfiles. In this section we discuss the former. Script mfiles will be discussed in a later section.

Function mfiles are preferred when the expression for the function is complicated, or we want to save the function to use in another session.

Suppose we need to compute the values of the function

$$f(x) = x \exp(-\sin(x))/(1+x^2).$$

We can create a function mfile, called `f.m`, so that to evaluate f at $x = 2$, we need only type the command `f(2)`. The mfile is a file that should be placed in the same directory where you are using MATLAB. Here is what the mfile looks like.

```
function y = f(x)
y = x*exp(-sin(x))/(1+x^2);
```

Written this way the function can only take scalars for x. However, if we write it using the symbols for the array operations, like this,

```
function y = f(x)
y = x.*exp(-sin(x))./(1+x.^2);
```

the function is now array-smart and can be used on vectors and matrices. Notice in the denominator we are adding the scalar `1` to the vector `x.^2` to produce another vector, which then divides in array fashion the factor `x.*exp(-sin(x))` .

Functions which are defined piecewise may also be constructed in an array-smart fashion. Consider the example

$$f(x) = \begin{cases} x & x < 0 \\ x^2 & 0 \le x < 2 \\ 4 & x \ge 2 \end{cases} .$$

The building blocks for this kind of function are the *characteristic* functions for intervals of the form $(-\infty, a)$ and (a, ∞). An mfile for this kind of function would be

```
function y = c(x)
y = (x < 3);
```

Check that $c(x) = 1$ for $x < 3$ and $c(x) = 0$ for $x \ge 3$. Now we make an mfile for f which is array-smart as follows:

```
function y = f(x)
y1 = x.*(x < 0);
y2 = x.^2.*( (x < 2) - (x < 0) );
y3 = 4*(1 - (x < 2));
y = y1 + y2 + y3;
```

Functions of several variables can also be defined in mfiles. We shall consider this topic in the next chapter.

Finally we note that the variables used in the mfile to define the function are "dummy" variables. One can use any variable names to call the function. For example for the function f defined above we can use the statements

```
s = -2:.1:4;
r = f(s);
```

The first command defines the vector **s** with 61 components, and the second command computes another vector **r** with $r_i = f(s_i)$ for $i = 1, \ldots, 61$.

1.6 Two dimensional graphs

MATLAB has an excellent set of graphic tools. In this section we will only touch on some of the most elementary ones. We begin with two dimensional graphs. MATLAB takes two vectors $\mathbf{x} = (x_1, \ldots, x_N)$ and $\mathbf{y} = (y_1, \ldots, y_N)$, locates the points (x_j, y_j) and joins them by straight lines. The command is `plot(x,y)`. The vectors $\mathbf{x} = (1, 2, 3, 4, 5)$ and $\mathbf{y} = (2, 3, 1, 5, -1)$ plotted this way produce the picture shown in Figure 1.1.

```
>> x = [1 2 3 4 5];
>> y = [2 3 1 5 -1];
>> plot(x,y)
```

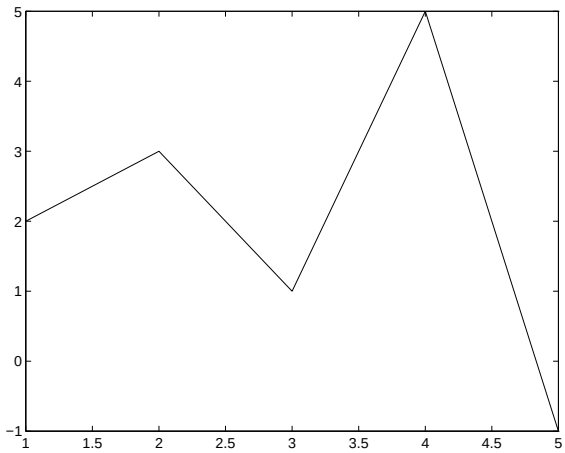

Figure 1: Plot **x** versus **y** for the vectors $\mathbf{x} = (1, 2, 3, 4, 5)$ and $\mathbf{y} = (2, 3, 1, 5, -1)$.

The circle of radius 2, with center at $(1, 3)$, is produced by the commands

```
>> theta = linspace(0, 2*pi, 51)
>> x = 1 + 2*cos(theta);
>> y = 3 + 2*sin(theta);
>> plot(x,y)
```

To plot the graph of a function like $\sin x$ on the interval $[0, 2\pi]$ we use the commands

```
>> x = linspace(0, 2*pi, 51)
>> y = sin(x);
>> plot(x,y)
```

The last two commands can be combined in one: `plot(x,sin(x))`. If the function f is defined by an array-smart mfile `f.m`, or as an inline function, we can also plot it with the command `plot(x,f(x))`.

The color of a single curve is by default blue, but other colors are possible. The desired color is indicated by a third argument which is a character string. For example, red is selected by `plot(x,y,'r')`. Note the single quotes around r. The color table is

y	yellow
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
b	black

For a complete listing of the combinations of colors and symbols, enter `help plot`.

There are two ways that we can plot several curves on the same graph. Remember a curve is determined by a pair of vectors $\mathbf{x}, \mathbf{y}$ of the same length n. Suppose there is another pair of vectors $\mathbf{z}, \mathbf{w}$ of the same length m where m may differ from n. The first way to plot the two curves on the same graph is with the command

```
>> plot(x,y,z,w)
```

In MATLAB4.2 the first curve will be in yellow, the second in magenta. In MATLAB5.0, the colors will be blue and green.

Two functions f and g given by array-smart mfiles `f.m` and `g.m`, or as inline functions, can be plotted on $[-1, 4]$ together with $\exp(x)$ by the commands

```
>> x = -1:.1:4;
>> plot(x,f(x),x,g(x),x,exp(x))
```

The three curves will be in different colors. The second way to plot several curves on the same graph uses the command `hold on`.

```
>> plot(x,y)
>> hold on
>> plot(z,w)
>> hold off
```

Both curves will now be same color. The three functions $f(x)$, $g(x)$, and $\exp(x)$ are plotted together these commands.

```
>> plot(x,f(x))
>> hold on
>> plot(x,g(x))
>> plot(x,exp(x))
>> hold off
```

A second, and very quick way, to graph functions is to use the command `ezplot`. It works this way. Suppose the function f is already prepared in a mfile `f.m`. Then to graph f on the interval, say, [1,5], we can do this in one command

```
>> ezplot('f',[1,5])
```

We can equally well make the command `ezplot('f',1,5)`. Note the single quotes around the `f` in the call. If f is defined as an inline function, the call is `ezplot(f,[1,5])`, *without* the single quotes. The `ezplot` command picks its own points for graphing, using more where the function changes rapidly, and fewer where it changes more slowly. We shall see later how `ezplot` can also be used to graph functions defined symbolically.

Labels and a title can be attached to the graph with addtional commands, for example

```
>> xlabel(' t, time after lift off,
                                        in seconds ')
>> ylabel(' h, height above ground in meters   ')
>> title(' vertical climb of rocket   ')
```

Further two dimensional graphing features

The `axis` command. When we use the command `plot(x,y)`, MATLAB automatically plots the curve on the rectangle $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$. If we wish to change this scale, perhaps to expand a portion of the graph, and instead plot on the rectangle $[a, b] \times [c, d]$, we follow the plot command with `axis([a b c`

`d])`. You can return the axis scaling to the automatic, default, mode with the command `axis('auto')` (alternate form `axis auto`).

The `zoom` command. This is another way to blow up a portion of the graph, using the mouse. Enter the command `zoom on`. Then move the pointer to the region of the graph you want to enlarge. Click with the left mouse button. This will blow up the size by a factor of two. Clicking again blows it up again by a factor of two. Clicking with the right mouse button has the opposite effect. The command `zoom out` restores the original figure. `zoom off` turns off the zoom feature.

The `ginput` command. This feature allows one to pick off the coordinates of points on a figure using the mouse. The command is `[x,y] = ginput(n)`. Move the pointer to n different points in the figure, and click at each point with the left mouse button. When you have clicked on the last point, the coordinates of the n points are displayed on the screen in column vectors **x** and **y**.

2 Basic MATLAB Continued

We begin with a short listing of some very helpful commands. Then we give a description of script mfiles, followed by a discussion of 3D graphics. We finish the chapter with instructions on how to save work, print out figures, and prepare documents.

2.1 Some very useful commands

Now that you can solve some equations and graph some functions, you will find the following utility commands very useful.

If you know the name of the command or feature, and want information about it, enter `help` *command name*. If you want to see the code of this command displayed on the screen, enter `type` *command name*. For example, the MATLAB feature `fzero` finds the zeros of a function of one variable. For information on how to use it, we enter `help fzero`. To see the code, we enter `type fzero`. To find where in the structure of directories `fzero` can be found, we enter `which fzero`.

Some of the help files and codes are rather long, and they go by on the screen very quickly. To see them one screen at a time, enter `more on` before entering any of the query commands. When you are done, enter `more off`.

All this information is very accessible if you know the name of the command. However, suppose you want to know if MATLAB has a command, or several commands, that deal with a certain kind of problem. In this case we use the command `lookfor`. For example, if we want to find if MATLAB has a function that finds the largest element of a vector or matrix, we might enter `lookfor largest`. The professional version yields the following listing.

```
>> lookfor largest

REALMAX Largest positive floating point number.
MAX    Largest component.
NNFMC Find largest column vector in matrix.
```

When we enter `help max` we find

```
>> help max

 MAX    Largest component.
    For vectors, MAX(X) is the largest element in X. For
    matrices, MAX(X) is a row vector containing the maximum
    element from each column. For N-D arrays, MAX(X) operates
```

```
along the first non-singleton dimension.

[Y,I] = MAX(X) returns the indices of the maximum values
in vector I.  If the values along the first non-singleton
dimension contain more than one maximal element, the index
 of the first one is returned.

MAX(X,Y) returns an array the same size as X and Y with the
largest elements taken from X or Y.  Either one can be a
 scalar.

[Y,I] = MAX(X,[],DIM) operates along the dimension DIM.

When complex, the magnitude MAX(ABS(X)) is used.  NaN's are
ignored when computing the maximum.

Example: If X = [2 8 4   then max(X,[],1) is [7 8 9],
                 7 3 9]

    max(X,[],2) is [8    and max(X,5) is [5 8 5
                    9],                  7 5 9].

See also MIN, MEDIAN, MEAN, SORT.
```

2.2 Script mfiles and programs

Script mfiles are used to collect a sequence of commands that may be lengthy, or tedious to type over and over again. Calling the name of the script mfile tells MATLAB to execute the sequence of commands, which constitute a kind of program. Here are several examples.

Example 2.1

Suppose that we wish to plot the functions $f_n(x) = x^n \exp(-nx)$ on the interval $[0, 20]$ for $n = 1, \ldots, 10$ on the same graph. We could do this by using the `plot` command and `hold on` over and over again on the command line. However a better way, which allows us to reproduce the graphs any time, is to write a short program, call it `graphs.m`, to do this. We shall use the notion of a *for loop*. Here is the script.

```
x = 0:.1:20;
axes
hold on
```

```
for n = 1:10
  plot(x, x.^n.*exp(-n*x))
end
hold off
```

The command `axes` sets up a figure window, and the command `hold on` will ensure that all the curves are plotted in the same figure. To run this script, enter the command `graphs` on the command line.

Example 2.2

Suppose the problem is to compute the terms of a Fibonacci sequence which is defined recursively by the formula

$$a_{n+1} = a_n + a_{n-1} \quad n = 1, 2, \ldots .$$

We must provide the starting values a_0 and a_1. We must also specify how many terms we want to compute. To compute the first 100 terms, with starting values $a_0 = 1,\ a_1 = -2$, we could use the following script mfile, call it `Fibo.m`.

```
a0 = 1
a1 = -2
for n = 2:100
  a = a1 + a0
  a0 = a1;
  a1 = a;
end
```

This program is not very flexible, because if we wish to change the starting values or the number of terms to be computed, we must edit the program. We can provide this information from the command line when we run the program by adding input statements. Here is the new, improved, program.

```
N = input('enter the number of terms to be computed')
a0 = input(' enter the value of a0  ')
a1 = input(' enter the value of a1  ')
for n = 2:N
   a = a1 + a0
   a0 = a1;
   a1 = a;
end
```

Now when we enter the command `Fibo`, the program will ask for the appropriate input, which we then enter. After the third piece of input is entered, the program will execute.

Script mfiles can call function mfiles or other script mfiles. In the next example, we give a crude implementation of Newton's method for finding a solution x_* of an equation $f(x) = 0$.

Example 2.3

Recall that Newton's method generates a sequence of approximate solutions according to the rule

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

We must provide a first guess x_0, and then can run to any number of iterations. Here is our code, `Newton.m`. It requires two auxiliary function mfiles; one for the function `f.m`, and another for the derivative, `df.m`.

```
N = input('enter the number of iterations   ')
x0 = input('enter the first guess for the root  ')
for n = 1:N
   x1 = x0 - f(x0)/df(x0)
   x0 = x1;
end
```

This program is unsatisfactory in that we do not know in advance how many iterations we will need to achieve a desired accuracy. From numerical analysis we know that we can estimate the error $|x_n - x_*|$ by the difference $|x_{n+1} - x_n|$. If we specify a given tolerance, say 10^{-6}, in advance, we want the iterations to continue until $|x_{n+1} - x_n| \leq 10^{-6}$. We do not specify the number of iterations in advance. The device of a *while loop* is designed to handle this situation. It tests the difference between two succesive iterates each time it enters the loop. If the difference $|x_{n+1} - x_n| > 10^{-6}$, the loop is entered; if not the program is terminated. Here is the new program `Newton.m`

```
tol = input(' enter the desired tolerance   ')
x0 = input(' enter the first guess for the root  ')
x1 = x0 - f(x0)/df(x0);
while abs(x1 - x0) > tol
   x0 = x1;
   x1 = x0 - f(x0)/df(x0)
end
```

To see how the convergence is progressing, it would be of interest to display the number of each iteration and the value of $f(x_n)$. We add a few more lines to program `Newton.m`

```
tol = input(' enter the desired tolerance   ')
x0 = input('  enter the first guess for the root   ')
x1 = x0 - f(x0)/df(x0);
disp('      n           x           f(x) ')
[0 x0 f(x0)]
[1 x1 f(x1)]
n = 1;
while abs(x1 - x0) > tol
   n = n+1;
   x0 = x1;
   x1 = x0 - f(x0)/df(x0);
   [n  x1  f(x1) ]
end
```

It would also be wise to put in a condition to limit the number of iterations, in case the iterations do not converge. This can happen when a bad first guess is made. If we wish to stop, after say 10 iterations, we can modify the while statement as follows:

```
while  abs(x1 - x0)*(n < 11) > tol
```

Recall that the function `(x < 11)` will be one when the inequality is satisfied, and zero when $x \geq 11$. Thus when n reaches 11, `abs(x1-x0)*(n < 11)` is zero, and the loop will not be reentered.

2.3 Operators on functions

MATLAB has a number of routines that operate on functions, called *function functions*. These routines generally have function names as well as variables as arguments. We give only a couple of examples that we shall use later.

The root finder `fzero` finds numerical estimates of the roots of an equation $f(x) = 0$. First we define f in an mfile, or as as inline function. If f changes sign in the interval $[x_0, x_1]$, then then there must be a root x_* of $f(x) = 0$ in this interval. When f is defined as an inline function, we can get a numerical estimate of the root with the call `root = fzero(f, [x0, x1])` . If f is defined in an mfile, the call is `root = fzero('f', [x0, x1])`. Note that in the latter case, we use single quotes around `f`. There are many options that can be used with `fzero`. In particular, it is possible to set the desired tolerance ε such that the

computed root $\tilde{x}$ satisfies $|x_* - \tilde{x}| \leq \varepsilon$. Information is available on `fzero` online with the command `help fzero`.

A second important routine that we shall use is a numerical integrator. If $f(x)$ is given on the interval $[a, b]$, the call `quad8(f, a,b)` makes a numerical estimate of $\int_a^b f(x)dx$. Again, when f is defined in an mfile, we must use single quotes in the call. Information is available on line with `help quad8`.

2.4 Functions of 2 variables

Functions of 2 and 3 variables can be defined in mfiles in the same way as functions of one variable. To make the function array-smart remember to use `.*`, `.^`, and `./`.

Example

Let $f(x, y) = \exp(-(x-3)^2 - (x-2)^2)$. The mfile for f is

```
function z = f(x,y)
   z = exp(-( (x-3).^2 + (y-2).^2 );
```

To define f as an inline function, we must specify the variables.

```
>> f = inline('exp(-(x-3).^3 - (x-2).^2', 'x', 'y')
```

Plotting functions of two variables

In this section we will see how to graph functions of two variables. The basic plotting variable for 2-D graphs is the vector. In 3-D graphs, the basic plotting variable is the matrix.

Most often we want to plot a function $f(x, y)$ over the rectangle $a \leq x \leq b, c \leq y \leq d$. First construct a mesh over the rectangle by selecting a stepsize in the x direction, Δx, and a stepsize in the y direction, Δy. Then construct the vectors `x` and `y` with the commands `x = a:delx:b` and `y = c:dely:d`. The next command creates two matrices : `[X,Y] = meshgrid(x,y)`. If n is the length of `x` and m is the length of `y`, both X and Y are $m \times n$ matrices. The m rows of X are all equal to the vector `x`, and the n colums of Y are all equal to the vector `y`. A corresponding $m \times n$ matrix of the values of f at the grid points is generated by the command `Z = f(X,Y)`. We assume that $f(x, y)$ is expressed by an array-smart mfile `f.m`. A three-dimensional wire mesh surface is generated by the command `mesh(X,Y,Z)` while a faceted surface is generated by the command `surf(X,Y,Z)`. We could, of course, combine two commands into one as `mesh(X,Y,f(X,Y))` or `surf(X,Y,f(X,Y))`.

We have written an mfile for the function $f(x, y) = \exp(-(x-3)^3 + (y-2)^2)$. To graph f over the rectangle $[0, 4] \times [0, 6]$ use the sequence of commands

```
>> x = 0:.1:4;
>> y = 0:.1:6;
>> [X,Y] = meshgrid(x,y);
>> Z = f(X,Y);
>> mesh(X,Y,Z)
```

Follow this with the command `surf(X,Y,Z)` to see the faceted surface. For another example, try `Z = sin(X-Y)`. The wire mesh surface looks like this (see Figure 2).

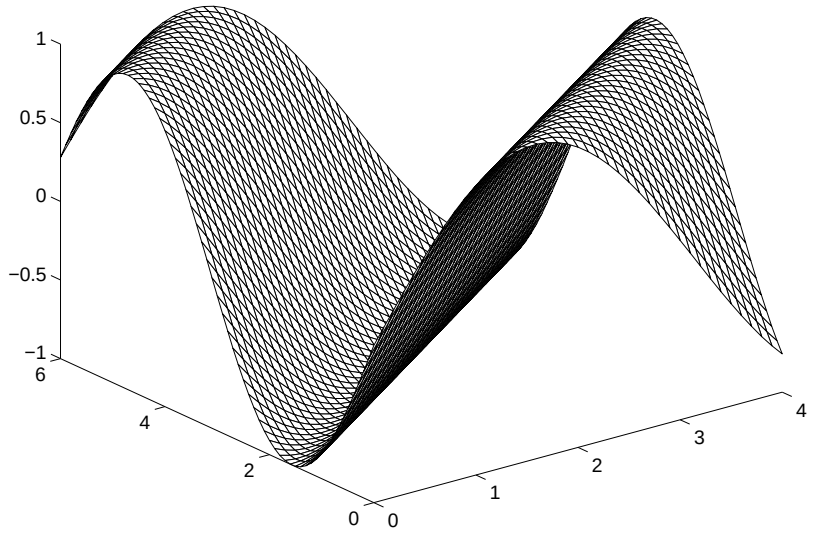

Figure 2: Mesh surface of $z = \sin(x - y)$.

2.5 MATLAB documents

Saving your work

You may have to stop a MATLAB session before you have finished a project and you would like to keep the work you have done so far. The mfiles will be kept in your directory for your future use. But there may be expressions created on the command line that you wish to keep. This can be done with the command `save`. For example, suppose you have entered some large matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and a symbolic expression `f = a*x^2 + b*x +c` in the course of a computation. In the next session, you do not wish to reenter these matrices or to retype the symbolic expressions. Instead you can save them to a file, e.g. `hotstuff`, with the

command `save hotsuff`. This will save the values of all the variables you have used. If you only want to save the values of $\mathbf{A}, \mathbf{B}, \mathbf{C}$, we can refine the command to `save hotstuff A B C`. At your next session, to retrieve these variables, use the command `load hotstuff`.

Saving figures

In addition to saving variables, we also want to be able to save the graphs and figures produced by our work. To save the current figure in the form of a postscript file for printing use the command `print -dps Fig1`. This will produce a postscript file in your current directory, `Fig1.ps`. It can be printed out using whatever commands your system uses.

Preparing MATLAB documents

It is important to able to present your MATLAB work in a well organized, readable manner. Here are some instructions to help you do this. We illustrate with an example. Suppose problem 1 in some assignment asks you to sum the power series for e^x with 5 terms, compare with the MATLAB function `exp(x)`, and plot the results on the interval $[-2, 2]$. This would be done with an mfile, which we shall call `myexp.m`. It would consist of the following sequence of commands.

```
x = -2:.2:2;
term = 1;
k = 1;
y = ones(size(x));
for n = 1:5
   term = term.*(x/k);
   y = y + term;
   k = k+1;
end
[x', y', exp(x)', (y-exp(x))']
maxerror = max(abs(y - exp(x)))
plot(x,y,x,exp(x), '--')
```

Now when you enter the command `myexp`, you will produce four columns of numbers on the screen, the number "maxerror" on the screen, and a graph in a figure window. To record this program to be turned in, together with the output, we use the `diary` commands. The command `diary` *file name* prepares all the following output, together with any keyboard commands, to be put in a text file that can be edited. The command `diary off` after running the program will actually write into the file. In our case, we would enter the commands

```
>> diary problem1
>> myexp
>> diary off
```

The file `problem1` contains the numerical screen output, but not the graph. It looks like this.

```
>> myexp
ans =
   -2.0000    0.3333    0.1353    0.1980
   -1.8000    0.2854    0.1653    0.1201
   -1.6000    0.2704    0.2019    0.0685
       .          .         .         .
       .          .         .         .
       .          .         .         .
    1.6000    4.8357    4.9530   -0.1173
    1.8000    5.8294    6.0496   -0.2202
    2.0000    7.0000    7.3891   -0.3891


maxerror =
    0.3891
>> diary off
```

Notice that the commands of the program itself are not put into the file `problem1`. To include the program commands as well, use the command `echo` in the sequence

```
>> diary problem1
>> echo
>> myexp
>> diary off
```

This has the disadvantage that the commands are repeated in the file `problem1` with each pass through the loop. For this example, you should cut and paste the file `myexp.m` into the beginning of the file `problem1`.

By editing the file `problem1`, it is now possible to add labels at the tops of the columns, and to add interpretive comments about the results of the calculations. Comments about the graphs can also be added here, with reference to Figure 1, Figure 2, etc. Here is the file `problem1`, after editing, with the program inserted at the beginning, and a second page for the graph.

```
                    Problem 1
    This is the program "myexp" used to compute a 5 term
approximation to the exponential function on the interval [-2,2].
     x = -2:.2: 2;
     term = 1;
     k = 1;
     y =  ones(size(x));
     for n = 1:4
        term = term.*(x/k);
        y = y+term;
        k = k+1;
     end
     [x', y', exp(x)', (y-exp(x))']
     maxerror = max(abs(y-exp(x)))
     plot(x,y,x,exp(x), '--')
     title('Figure 1. 5 term approx,
                            and true exp(x) (dashed line)')

    The values of the approximation are put in the vector y, and
compared with the MATLAB  exponential. Here are the results.

       x         y       exp(x)    error = y - exp(x)

   -2.0000    0.3333    0.1353    0.1980
   -1.8000    0.2854    0.1653    0.1201
   -1.6000    0.2704    0.2019    0.0685
      .         .         .         .
      .         .         .         .
      .         .         .         .
    1.6000    4.8357    4.9530   -0.1173
    1.8000    5.8294    6.0496   -0.2202
    2.0000    7.0000    7.3891   -0.3891


maxerror =
    0.3891

    Comments: As we can see from Figure 3 (attached), the
5 term approximation does quite well in the interval [-1, 1].
In fact, from the table, we can see the maximum error over
```

```
this interval is .0099.
```

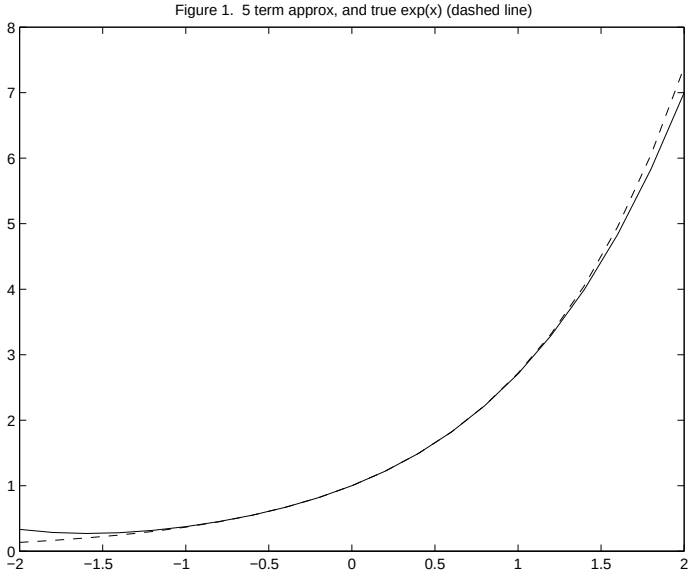

Figure 3: Figure produced by the script mfile `myexp`.

3 More Features of MATLAB

3.1 The command feval

The command `feval` is used to take the name of a function in the form of string, call the function, and evaluate it. For example, if $f(x, y) = x^2 - 2y$ is defined as an inline function,

```
>> f = inline('3*x.^2 -2*y', 'x', 'y');
        >> feval(f,2,3)
        ans =
            6
```

Of course, we could have more easily written `f(2,3)`. If the same function f is given in an mfile `f.m`

```
function z = f(x,y)
   z = 3*x.^2 - 2*y;
```

then we must use the name of f in the form of a string as the argument of `feval`.

```
>> feval('f', 2,3)
ans =
    6
```

The importance of `feval` comes in writing function mfiles which take functions as arguments. For example, the MATLAB rootfinder `fzero` has the call

```
fzero(fname, [x1,x2])
```

where `fname` is a string which is the name of a function. `feval` is used in the code of `fzero` to evaluate the function. Thus when f is given as an inline function, we use `fzero(f,[x1,x2])` and when f is given in an mfile, we use `fzero('f',[x1,x2])`. This construction allows the function function `fzero` to accept any name for a function, f, g, h, F, G, etc. The same is true of the function mfiles such as `simp2` that are written for this text.

Example 3.1

The secant rule for finding the zero of a function $f(x)$ is similar to Newton's method but in place of the value of the derivative, it uses a difference quotient. It uses two starting values x_0, x_1 and the rule

$$x_{n+1} = x_n - f(x_n)\left[\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}\right].$$

The secant rule does not converge quite as fast as Newton's method, but has the advantage of using only the function values.

```
function out = Secant(f, x0,x1,tol)
  x2 = x1 -feval(f,x1)*(x1-x0)/(feval(f,x1) -feval(f,x0));
  disp('  n          x           f(x)      ')
  [0,x0,feval(f,x0)]
  [1,x1,feval(f,x1)]
  [2,x2,feval(f,x2)]

  n = 2
  while abs(x2-x1)*(n< 11) > tol
    x0 = x1;
    x1 = x2;
    x2 = x1 - feval(f,x1)*(x1-x0)/(feval(f,x1) - feval(f,x0));
    [n, x2, feval(f,x2)]
    n = n+1;
  end
```

3.2 Vectorizing computations

Computations in MATLAB that involve loops, or nested loops, can often be speeded up if we take advantage of the many array operations of MATLAB. Our goal in MATLAB programming is to eliminate as many loops as possible.

Example 3.2

The operation `sum` is applied to a vector, and adds up the components. Here are two codes to compute the sum $\sum_1^N n^p$. The long way is to do this in a loop.

```
function out = f(p,N)
   s = 0;
   for n = 1:N
      s = s + n^p;
   end
   out = s;
```

A more efficient way is to use the `sum operator` and the array operation `.^`.

```
function out = f(p,N)
   x = [1:N].^p;
   out = sum(x);
```

Example 3.3

The two-dimensional midpoint rule for integration over a rectangle R is

$$\int\int_R f \; dxdy \approx \sum_{j=1}^{n}\sum_{i=1}^{m} f(x_j, y_i)\Delta x \Delta y$$

where (x_j, y_i) is the center of the subrectangle $R_{i,j}$ with sides Δx and Δy. The long way to implement this rule is to use nested loops:

```
function out = midpt(f,corners,n,m)
     a = corners(1); b = corners(2);
     c = corners(3); d = corners(4);
     delx = (b-a)/n; dely = (d-c)/m;
     x = linspace(a+.5*delx, b-.5*delx,n);
     y = linspace(c+.5*dely, d-.5*dely,m);
     s = 0;
     for i = 1:m
        for j = 1:n
           s = s + feval(f,x(j),y(i));
        end
     end
     out = s*delx*dely;
```

The double loop can be replaced with vector, matrix operations which are more efficient. The operation `sum`, when applied to a matrix, sums down each column and puts the sum of each column in a row vector. Then applying `sum` a second time sums the elements in this row vector, yielding the sum over all the elements of the matrix.

```
[X,Y] = meshgrid(x,y);
F = feval(f,X,Y);
out = sum(sum(F))*delx*dely;
```

`F` is the matrix of function values $f(x_j, y_i)$.

3.3 Programming logic

The programs we have written so far involved only simple for loops or a while loop. Often, however, a program must make a choice and proceed to make different calculations, depending on a parameter. The relevant commands are `if`, `else`, and `elseif` .

Example 3.4

Consider an income tax system with two brackets. For income less than $ 20,000, the rate is 10%, and for income in excess of $ 20,000, the rate is 15%. A short program to compute the tax could be written as follows.

```
function out = tax(income)
      if income < 20000
         out = .1*income;
      else
         out = 2000 + .15*(income - 20000);
      end
```

Notice that the `end` command is needed to close the `if`, `else` sequence.

Now suppose the system has a third bracket, with a rate of 20% for income inexcess of $ 50,000. We modify the function mfile `tax` as follows.

```
function out = tax(income)
      if income < 20000
         out = .1*income;
      elseif 20000 <= income < 50000
         out = 2000 + .15*(income-20000);
      else
         out = 6500 + .2*(income - 50000);
      end
```

One can add any number of branches with more `elseif` statements.

Another useful logical command is the `break` command which terminates a loop when a certain value is reached in a computation before the index of the loop is exhausted.

Example 3.5

Return to our example of computing the terms of a Fibonacci sequence, Example 2.2. Suppose we wish to stop the computation whenever $|a_j| > 1000$. The modified script is

```
N = input('enter the number of terms to be computed  ')
a0 = input('enter a0  ')
a1 = input('enter a1  ')
for n = 2:N
   a = a1+a0
   if abs(a) > 1000
```

```
        break
    end
    a0 = a1; a1 = a;
end
```

For descriptions of other logical commands enter `help or` or `help and`.