Exerpt from *A MATLAB Companion for Multivariable Calculus* by Jeffery Cooper, Harcourt/Academic Press, 2001.

1. The Command Line

In this chapter, we discuss operations that can be performed from the command line. In Chapter 2, we discuss mfiles and programs.

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 $(2 + 3.5^2 - 4 \cdot 7)/12$,

```
>> (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;
>> z = (x-y)/12;
>> z
z =
   -1.1458
```

MATLAB does 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`.

Error messages

If we enter an expression incorrectly, MATLAB will return an error message, which sometimes locates the error. For example, in the following, we left out the `*` in `3*x`.

```
>> x = 4;
>> 3x
??? 3
    |
Missing operator, comma, or semicolon.
```

Another example.

```
>> 2*(x+y
??? 2*(x+y
          |
A closing right parenthesis is missing.
Check for a missing ")" or a missing operator.
```

Making corrections

To make corrections, we can, of course, retype the expression. But if the expression is lengthy, we may make more mistakes by typing a second time. Unfortunately we can not move the cursor to the line we wish to repair. Instead we can press the up arrow key until we reach the desired line and then the left and right arrows, until we reach the offending characters. Type in the correction and enter return.

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.

HELP ! ! Help with most operations is available with a keystroke, thanks to the on line help provided by MATLAB. To get information on a particular command or operation, simply enter `help` *command name*. For example, to get information on how to use the plotting commands, enter `help plot`.

1.2 Vectors and matrices

Vectors and matrices are the basic elements of the MATLAB environment. In this text we shall be using the word *vector* in two, related, ways.

In Chapter 3, we shall speak of vectors as directed line segments in two and three dimensional space, used to represent physical and geometric quantities such as force and velocity.

In this chapter, we shall use vector to mean an ordered list of numbers, written either horizontally, or vertically. For example,

$$\mathbf{u} = [2, 1.3, \sqrt{2}, 8, -4, \pi]$$

or

$$\mathbf{v} = \begin{bmatrix} 1 \\ -2 \\ \pi \\ 4.2 \end{bmatrix}.$$

We say that **u** is a row vector and that **v** is a column vector.

A *matrix* is a rectangular array of numbers. For example,

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 9 \\ 4 & 5 & 6.1 & -2 \\ \pi/2 & 1/3 & 4 & -1 \end{bmatrix}.$$

The dimensions of a matrix are the number of rows and the number of columns, with the number of rows usually given first. The matrix **A** above is a 3×4 matrix. The row vector **u** is a 1×6 matrix, and the column vector **v** is a 4×1 matrix. A single number, like 5.2, is a *scalar* and can be considered a 1×1 matrix. The entries in a matrix often are written $a_{i,j}$ with i being the row index and j being the column index. For example, in the matrix **A**, above, $a_{2,1} = 4$ and $a_{3,2} = 1/3$.

The *transpose* of an $m \times n$ real matrix **A** is the $n \times m$ matrix that results from interchanging the rows and columns of **A**. The transpose matrix is denoted $\mathbf{A}^T$. The transpose of the matrix **A** above is

$$\mathbf{A}^T = \begin{bmatrix} 1 & 4 & \pi/2 \\ 2 & 5 & 1/3 \\ 3 & 6.1 & 4 \\ 9 & -2 & -1 \end{bmatrix}.$$

Various operations can be performed on vectors and matrices and we shall illustrate them in the context of MATLAB.

Forming vectors and matrices

Matrices can be entered by typing in the elements one at a time. To enter the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix},$$

we type

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

A =
   1     2     3
   4     5     6
```

Notice that we use a semicolon to separate the rows. 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.

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, the command `A'` produces the Hermitian 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.'`. That is, put a dot before the apostrophe.

To determine the dimensions of a vector or matrix, use the command `size` as follows:

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

>> size(x)
ans =
     1     5

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

We can view a particular element in a vector or matrix by specifying its location:

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

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

Often we must deal with vectors or matrices which are too large to enter one element at a time. If there is some formula or some regular pattern to the elements, we may be able to enter them as follows. Suppose we want to enter a vector **`x`** consisting of points $(0, .1, .2, .3, .4, \ldots, 5.9, 6)$. 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 dimensions as a given 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 Array operations

Arithmetic of matrices

There is an obvious, natural, way to add and subtract matrices.

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

Usually, we can add together only matrices having the same dimension. There is an exception in MATLAB, however, which is very useful. Suppose we want to add the same number c to each element of a matrix $\mathbf{A}$. This can be done with the command `A + c*ones(size(A))`, or more simply, `A` $+$ `c`. In particular, if $\mathbf{x}$ is a vector, we can add a scalar t to each component of $\mathbf{x}$ with the command `x+t`.

We can always multiply a matrix by a scalar, or divide by a nonzero 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 **A** and **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 **C** of the same size with elements $c_{i,j} = a_{i,j}b_{i,j}$. For example using the same 2×3 matrices **A** and **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 2×3 matrix

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

and

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

a 3×1 column vector. We define the product $\mathbf{Ax}$ 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 definition 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

$$\mathbf{Ax} = \mathbf{b}. \tag{0.1}$$

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 $\mathbf{B}_j = [b_{i,j}],\ j = 1, \ldots, p$. We define

$$\mathbf{AB} = \mathbf{C} \tag{0.2}$$

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. Remember we use 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 $\mathbf{AA} \ldots \mathbf{A}$. 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 $\mathbf{Ax} = \mathbf{b}$ 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

MATLAB basically has two kinds of functions, numerical functions and symbolic expressions of functions. A *numerical function* is really a short program that operates on numbers to produce numbers. A *symbolic expression of a function* operates on symbolic variables to produce symbolic results. These symbolic expressions can be manipulated with operations like differentiation and integration. We shall discuss symbolic expressions of functions in the next section.

MATLAB has the usual built in numerical 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**.

Nevertheless, we will often need to build our own numerical 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 a later chapter.

Prior to version 5.0 of MATLAB numerical functions could only be constructed in separate files called mfiles. This way of constructing functions will be covered in Chapter 2.

Now with versions 5.0 and higher, there is an easy way of constructing a numerical function on the command line. This kind of numerical function 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')
```

Here we have used the symbol `.^` for the array operation. Functions created this way can accept 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
```

We shall need our numerical functions to be array-smart to do many computations, and for the purposes of graphing.

One of the most common mistakes of beginners is to forget to make their numerical functions array-smart by inserting the dot before the operations `*`, `/`, and `^`.

Unfortunately, we cannot add or multiply inline functions to produce a new function. If we define the inline function g by the command

```
>> g = inline('cos(x) + x')
```

we *cannot* use the command

```
>> h = f+g
```

to produce the function $f + g$. Instead we must define a new inline function

```
>> h = inline('x.^3 + 2*x - 1 + cos(x)')
```

1.6 Symbolic calculations

Up to this point, we have been using MATLAB on your computer as a large, sophisticated calculator. For example, if we enter a matrix **A** of numbers, we can find its determinant as a number. We have also created numerical functions. However, MATLAB also has the capability to manipulate expressions symbolically. There are tools to perform algebraic operations, differentiate and integrate functions, solve systems of equations, and solve ordinary differential equations. These tools come from the software program Maple developed at the University of Waterloo, Canada.

Creating symbolic expressions

Variables x, y, z, a, b, c, etc. can be declared symbolic variables with the command

```
>> syms x y z a b c
```

This command is a short cut for the more elaborate command `sym('x', 'y', 'z', 'a', 'b', 'c')`, or even more deliberately, `x = sym('x')`, `y = sym('y'), . . . .` We can then define expressions using these variables and these expressions can be manipulated symbolically. For example a matrix **A** can be defined by

```
>> A = [ a b 1; 0 1 c; x 0 0 ]
A =
[a, b, 1]
[0, 1, c]
[x, 0, 0]
```

Since `A` is a symbolic expression, we can calculate its determinant in terms of the variables a, b, c, x with the usual MATLAB command

```
>> d = det(A)
d = x*(b*c-1)
```

Functions defined symbolically

A function $f(x)$ can be defined in terms of a symbolic expression by this kind of command.

```
>> f = a*x^2 + b*x +c + 2*cos(x)
```

Notice that we do not use the array operations `.^, .*, ./` in symbolic expressions because symbolic expressions are not applied directly to vectors and matrices.

The symbolic expression for this function cannot be evaluated with the simple command `f(2)`. We will need another set of commands which are explained a bit further on.

Now we can differentiate this symbolic expression with the command (and output)

```
>> diff(f)
ans = 2*a*x+b-2*sin(x)
```

MATLAB differentiates with respect to the variable closest to x in the alphabet. If we wish to differentiate f with respect to the variable a, we must specify that in the command: `diff(f,a)`. If we wish to make further operations on the derivative we can give it a name, which will be the name for another symbolic expression:

```
>> fprime = diff(f)
fprime = 2*a*x+b-2*sin(x)
```

The second derivative can be computed by differentiating the expression `fprime`, or by using a variation on the diff operation,

```
>> diff(f,2)
>> 2*a-2*cos(x)
```

Higher derivatives are calculated with `diff(f,3)`, `diff(f,4)`, etc.

We can also find the antiderivative of functions defined symbolically. For example, using the same function defined above, we have

```
>> int(f)
ans =
1/3*a*x^3 + 1/2*b*x^2 +c*x +2*sin(x)
```

This operation provides us with an indefinite integral, to which we may add any constant. To compute the definite integral, over say $[0, 3]$, we use the command

```
>> int(f,0,3)
ans =
9*a +9/2*b +c*c+2*sin(3)
```

Here we assumed that we wanted to integrate the expression with respect to the variable x. If instead, we wanted to consider a as the variable of integration, we must specify that, with the command

```
int(f,a)
ans =
1/2*a^2*x^2 +b*x*a +c*a+2*cos(x)*a
```

Many other variations are possible. To see them enter `help sym/int.m`

Evaluating symbolic expressions

Next, how do we specify the values of the parameters in the expression, and how do we evaluate the symbolically defined function at a point? This is done using the substitution command `subs`. The syntax is `subs(f,old,new)` where the old values of the parameters and variables are replaced by new values.

For example, if we wish to evaluate the function f defined above at $x = 2$, leaving in the parameters a, b, c, we enter

```
>> subs(f,x,2)
ans =
9*a+3*b+c+2*cos(3)
```

The result is still a symbolic expression. If we wish to specify the values of the parameters, say $a = 2,\ b = -3,\ c = 9$, we do it this way:

```
>> g = subs(f, [a b c], [2 -3 9])
g =
2*x^2-3*x+9+2*cos(x)
```

Now we have a symbolic expression depending on the one variable x. To evaluate this function at a particular point, say $x = -1.5$, we can make another substitution `subs(g,x,-1.5)` with the answer of `18 + 2*cos(3/2)`. The result is still a symbolic quantity. If we wish to convert it to a floating point number in double precision, we use `double(18 + 2*cos(3/2))`, or in one command as `double(subs(g,x,-1.5))`. Again, many variations are possible. For further information, enter `help sym/subs.m`.

In Chapter 13, we discuss how to convert symbolic expressions to inline functions. This is important for graphing functions of several variables that arise in symbolic computations.

Solving equations symbolically

MATLAB can also solve certain equations symbolically, in terms of parameters in the equation. For example, to solve the equation $ax^2 + bx + c = 0$ we define the symbolic variables x, a, b, c and the expression $f = ax^2 + bx + c$ with commands

```
>> syms x a b c
>> f = a*x^2+b*x+c
>> solve(f)
ans =
[ 1/2/a*(-b+(b^2-4*a*c)^(1/2))]
[ 1/2/a*(-b-(b^2-4*a*c)^(1/2))]
```

Of course, these are the two solutions of the quadratic formula. The command `solve` assumes you want to solve the equation $f(x) = 0$.

For another example, consider the equation

$$\ln(y) - \ln(r - y) = kt + C.$$

To solve for y in terms of t, r, k and C, we can use the symbolic expression for $f = \ln(y) - \ln(r - y) - kt - C$.

```
>> syms t y r k C
>> f = log(y) - log(r-y) - k*t - C
>> y = solve(f,y)
y  =
r/(1+exp(k*t+C))*exp(k*t+C)
```

We can then find that value of t such that $y = 5$, in terms of the other parameters r, k, C with the commands

```
>> solve(y-5,t)
ans =
-(-log(5/(r-5))+C)/k
```

We shall investigate how to solve systems of equations involving several variables in later chapters.

1.7 Two dimensional graphs

Graphing numerical functions

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. The basic MATLAB graphing procedure in two dimensions is to take a vector of x coordinates, $\mathbf{x} = (x_1, \ldots, x_N)$, and a vector of y coordinates, $\mathbf{y} = (y_1, \ldots, y_N)$, locate the points (x_j, y_j), and then join them by straight lines. The command is `plot(x,y)`. The vectors $\mathbf{x} = (1, 2, 3, 4, 5)$ and $\mathbf{y} = (-1, 2, 3, 1, 5)$ plotted this way produce the picture shown in Figure 1.1.

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

We graph a numerical function in the same way. For example, to graph the function $\cos x$ on the interval $[-\pi, \pi]$, we first create a vector of x coordinates. Then we create a vector of y coordinates which are the values of $\cos x$ at these points. Finally, the points are plotted and joined by straight lines.

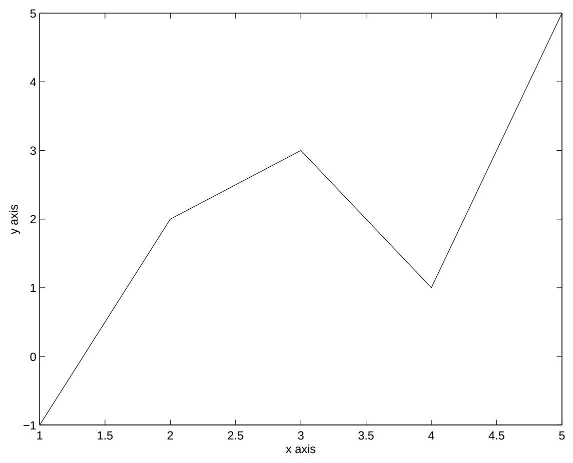

Figure 1: Plot $\mathbf{x}$ versus $\mathbf{y}$ for the vectors $\mathbf{x} = (1, 2, 3, 4, 5)$ and $\mathbf{y} = (-1, 2, 3, 1, 5)$.

```
>> x = linspace(-pi, pi, 51)
>> y = cos(x);
>> plot(x,y)
```

If the function f is defined as an inline function, we can graph it with the command `plot(x,f(x))`. For example, if we want to plot $f(x) = x^3 + x - 1$ on the interval $[1, 5]$, we use the commands

```
>> f = inline('x.^3 + x -1')
>> x = linspace(0,5, 101);
>> plot(x, f(x))
```

The color of a single curve, in MATLAB 5.0 or higher, 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
k	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 $\mathbf{x}, \mathbf{y}$ each with the same dimensions $n \times 1$ or $1 \times n$. Suppose there is another pair of vectors $\mathbf{z}, \mathbf{w}$ with dimensions $m \times 1$ or $1 \times m$ 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 and higher, the colors will be blue and green.

Two functions f and g given as array-smart 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 e^x are plotted together with these commands.

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

The ezplot command

The command `ezplot` is used primarily to graph functions which are defined symbolically. If f is defined by a symbolic expression and we wish to graph it on the interval $[1, 5]$, we can do it with the one command, `ezplot(f, [1,5])`. For example

```
>> syms x
>> f =  cos(x)^2*exp(x)
>> ezplot(f, [1,5])
```

This can be most useful after a symbolic calculation leads to a complicated expression. Using the function $f(x) = (\cos x)^2 \exp x$, if we want to quickly graph the second derivative of f, we could add the lines

```
>> g = diff(f,2)
>> ezplot(g, [1,5])
```

The `ezplot` command picks its own points for graphing, using more where the function changes rapidly, and fewer where it changes more slowly.

In versions 5.2 and higher of MATLAB the `ezplot` feature has been extended to graph curves given parametrically in two and three dimensions (with animation). It has also been extended to graph functions of two variables. We shall see these new features as each topic is considered.

Further graphing features

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

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

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 enlarge 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 blow up. Click with the left mouse button. This will enlarge the portion by a factor of two. Clicking again enlarges it 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.

1.8 Managing the workspace and getting help

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

Workspace commands

These commands allow you to find what you have in your workspace, and how to clear out unneeded variables.

`who` lists variables currently in the workspace and their type.

`clear` clears the workspace; all variables are removed.

`clear x y g` removes only the variables x, y and the function (either inline or symbolic) g.

`clf` clears the figure window.

`close` closes the figure window.

Getting information

Remember 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, suppose we want to know if MATLAB has a function that finds the largest element of a vector or matrix. We would 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.
                        .
                        .
                        .
See also MIN, MEDIAN, MEAN, SORT.
```

2. Beyond the Command Line

We discuss how to create and edit files in MATLAB. This is followed by a description of function mfiles and script mfiles. We finish the chapter with instructions on how to save work, print out figures, and prepare documents.

2.1 Creating and editing files in MATLAB

Working in MATLAB from the command line is virtually independent of the type of machine you are using. The different versions of MATLAB for PC, Mac, and Unix machines are adapted to run the same way on each of these platforms. However, when we venture beyond the command line, there are differences. We shall need to create and edit files, called *mfiles*, to

(i) create and save more complicated functions;

(ii) write and record longer sequences of commands.

PC's and Macs

On PC's and Mac's, MATLAB provides its own editor. In the upper left corner of the command window, click on the word "File". This opens the "File" menu. To write a new mfile, click on the line "New". This will bring up another window which is the MATLAB Editor/Debugger. After writing your file, usually a sequence of MATLAB commands, open the "File" menu of the Editor/Debugger window. Then you can name your file and save it with the "Save as" command. Usually, this will save your file in the current working directory and MATLAB will be able to find it when you call for it from the command line. However, if you are working on a shared system, there may be different arrangements and you must check with the system manager.

After you have saved your file, do not close the Editor/Debugger window. All too often, there is an error in the sequence of commands and you must return to the file to change it. With the file still in the Editor/Debugger window, you can make changes. However, these changes will not be recorded until you again go to the "File"menu of the Editor/Debugger window, and click on "Save".

Unix machines

Prior to version 5.2, Unix versions of MATLAB do not provide their own editor. In this case you must use your choice of Unix editor, such as `vi`, `emacs` or `pico`, in a separate window into the same working directory. It is possible to work from the MATLAB command window by entering the command `!vi [file name]`. In this case, MATLAB turns over control to the local system until you have finished editing the file.

With versions 5.2 and higher of MATLAB the command `edit` brings up the Editor/Debugger window, and you can use it as if you were working on a PC.

Now you may be impatiently asking, what kind of files will we be writing?

2.2 Mfiles

Mfiles are a very convenient, flexible way of collecting sequences of commands that may be lengthy or tedious to type over and over again. Mfiles may be saved to be used at another time. There are two kinds of mfiles: function mfiles and script mfiles. The names of mfiles always have the extension `.m`.

Function mfiles

Function mfiles are mostly used to write numerical functions whose expression is long or complicated and which we want to save for future use.

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 enter `f(2)` on the command line. 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 mfiles always begin with a function statement.

```
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, ∞). For example, the characteristic function for $(-\infty, a)$ is $c(x) = 1$ for $x < a$ and $c(x) = 0$ for $x \geq a$. We use the MATLAB logical expression `(x < a)`. When applied to a scalar x, this function returns a 1 if the inequality is true, and a 0 if it is false. When applied to an n vector `x` $= (x_1, \ldots, x_n)$, the logical function `(x < a)` returns an n vector of 0's and 1's, with a 1 whenever the inequality if true and a 0 whenever it is false. The logical functions `(x >a)` and `(x <=a)` work in the same way. An mfile for the characteristic function of the interval $(-\infty, 3)$ would be

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

Check that $c(x) = 1$ for $x < 3$ and $c(x) = 0$ for $x \geq 3$. 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;
```

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 $i = 1, \ldots, 61$.

Summary of function construction

We have now seen three ways to create functions with MATLAB.

Numerical functions are constructed using inline functions (Section 1.5) and function mfiles (this section).

Symbolic expressions for function are constructed, and manipulated, using the symbolic operations described in Section 1.6.

Graphing

The command `plot` works with numerical functions defined in mfiles exactly the same way it works with inline functions, e.g. `plot(x,f(x))` graphs the function given by the mfile `f.m`.

However, the command `ezplot` uses a slightly different call. Remember for a function given as an inline function, or defined symbolically, the call is `ezplot(f,1,3)`. When the function is given in an mfile `f.m`, the call is `ezplot('f', 1,3)`. Note the single quotes. We shall see this difference often.

`ezplot` is an example of a function mfile which can operate on other functions. These function mfiles have the *name* of a function as an argument in the call. When the function is given as an inline function, the name of the function is `f` or `g`, etc. When the function is given in an mfile, the name of the function is `'f'` or `'g'`, etc. We give two more examples of this type of function mfile in section 2.3.

2.3 Function functions

MATLAB has a number of routines that operate on functions, called *function functions.* These are function mfiles that 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 an inline function. If f is continuous and changes sign in the interval $[x_0, x_1]$, 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`.

Example 2.1

The function $f(x) = \sin x - x/2$ changes sign in the interval $[1, 3]$. To find the root of $f(x) = 0$ in this interval we use the following commands:

```
>> f = inline('sin(x) - x/2')
>> root = fzero(f, [1,3])
   root = 1.8955
```

There are many options that can be used with `fzero`. The function function `fzero` is discussed further in Chapter 7. See also the on line help.

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. We shall discuss this numerical integrator more in Chapter 9. Information is available on line with `help quad8`.

2.4 Script mfiles

Script mfiles are used to collect a sequence of commands that constitute a program. When we enter the name of the script mfile on the command line, the program will be executed. Here are two examples.

Example 2.2

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`, which performs this sequence of repeated operations. We shall use the notion of a *for loop.* Here is the script.

```
x = 0:.1:20;
for n = 1:10
  plot(x, x.^n.*exp(-n*x))
  hold on
end
hold off
```

The command `end` is needed to close the loop. To run this script, enter the command `graphs` on the command line. *Do not* enter the command `graphs.m`.

Example 2.3

In this example, we shall use the root finder `fzero` to find the four roots of the equation $f(x) = e^{-x} - \sin(x) = 0$ that lie in the interval $[0, 10]$. Here is a script that allows us to enter estimates for the four roots at run time, and then calculates the roots.

```
f = inline('exp(x) - sin(x)')
x = linspace(0, 10, 101);
plot(x, f(x), x, 0*x, 'g')
est = input('enter the 4 estimates
                        as a four vector [*,*,*,*]   ')
for n = 1:4
    root = fzero(f, est(n))
end
```

In plotting the graph, we also plotted the function identically equal to zero. This puts an x axis in green in the figure and makes it easier to see where the roots are located. After plotting the graph, the program waits for the user to enter four numbers in the form of a vector $[a, b, c, d]$.

Entering Comments

In a function mfile or a script mfile which involves several steps, it is very helpful for you, or for another reader, to identify the steps with comment lines. A *comment line* begins with the percent sign,`%`. When a script or function mfile is executed, the comment lines are ignored. For an example of the use of comment lines, see the script mfile `myexp.m` in section 2.5.

Workspace An important difference between script mfiles and function mfiles is in the way
the workspace is used. In a script mfile, all definitions of variables and calculations are made in a workspace which is accessible from the command line. In Example 2.3, the vector `x` can be viewed immediately after running the script simply by entering `x` on the command line.

By contrast, in a function mfile, the variables are not accessible from the command line. A function mfile has its own work space independent from the command line workspace. This arrangement allows one to use variable names in a function mfile that are the same as in other function mfiles with no question of confusion. For example, practically every function mfile using functions of one variable calls that variable x.

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 hotstuff`. 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

To save a figure so that you can do further work on it, put the commands that generated the figure in an mfile, e.g., `fig1.m`. When you enter `fig1` on the command line, the figure will be generated.

To print out a figure, click on the ”file” button on the upper left of the figure window and select print. You can also type `print` on the command line. This should print out the figure if you are working on a stand-alone machine connected to a printer. If you are working in a network of machines, you may need additional instructions. Ask your system manager for help.

To prepare a figure to be included in another document, give the figure a name, e.g., Fig1, and use the command `print -deps Fig1`. This will save the figure in the form of a Encapsulated Postscript file, `Fig1.eps`, that should be stored in your current working directory. The figure can also be printed out if your machine or system can print out postscript files. For a list of printing options, enter `help print`.

Preparing MATLAB documents

It is important to be 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 a script mfile, which we shall call `myexp.m`. It would consist of the following sequence of commands.

```
% define the vector of points where the function is to be
% computed and plotted.
x = -2:.2:2;

% the first term in the approx. is set equal to 1.
term = ones(size(x));
y = term;

% add up the terms and store the result in the vector y
for n = 1:4
   term = term.*(x/n);
   y = y + term;
end

% display the results as column vectors
[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 `type` in the sequence

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

The command `type` reproduces the code of any MATLAB mfile on the screen.

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 = ones(size(x));
     y =  term;
     for n = 1:4
        term = term.*(x/n);
        y = y+term;
     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 1 (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.
```

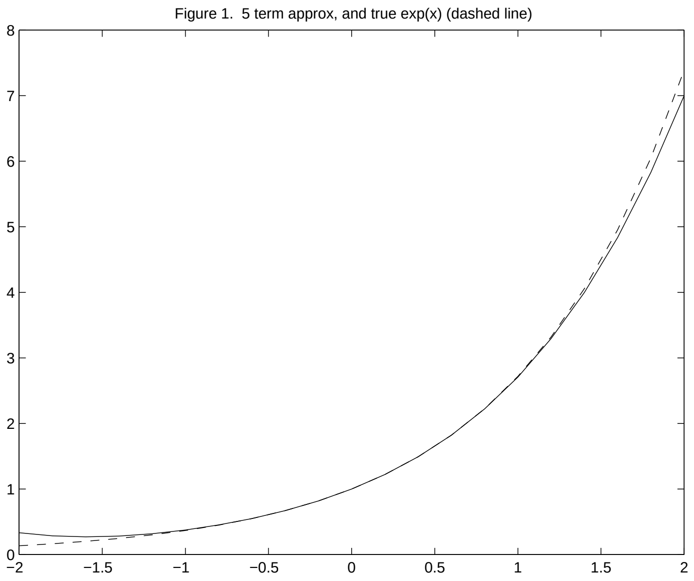

Figure 2: Figure produced by the script mfile `myexp`.

The diary command in Windows

If you are working on a PC with Windows (is there anything else?), put the diary into a file with the extension `txt`. For example

```
diary myexp.txt
```

A file with the extension `txt` can be immediately viewed and edited with the Notepad editor.