

2 *Basic Operations*

The first factories may well have been pottery works that operated in ancient Greece and Rome. Later, Roman factories manufactured glassware and bronzeware. In the Middle Ages, silk factories operated in Syria. England, France, and Italy hosted textile factories. By 1600, many factories existed in Western Europe, producing goods such as firearms, gunpowder, cast iron, glass, paper, clothing, soap, and beer.

Typically, early factories were merely large workshops where the laborers functioned independently, carrying out processes by means of hand labor and simple tools. Today's production processes, however, tend to be very complex, consisting of many interacting parts, all of which must work together effectively. Experience has shown that changes to one part of a system may actually decrease the effectiveness of the whole. The creation of a modern factory generally requires the talents of an industrial engineer. Industrial engineers analyze and design production systems. More specifically, they specify and integrate the components of people, machines, and facilities to create efficient and effective systems that produce goods and services. Generally, the criteria used in analyzing and designing production systems include productivity, quality, and cost.

In the Engineering Application example at the end of this chapter, you will see how an industrial engineer might use MATLAB to perform a simple economic analysis of a production system.

INTRODUCTION

This chapter describes the basic operations of MATLAB. Once you have grasped the concepts covered here, you will be ready to move on to Chapter 3, where you learn how to write basic programs. From there, you can take on the more advanced topics of the remaining chapters of this text. As stated in Chapter 1, MATLAB treats all numbers and sets of numbers, including scalars and vectors, as matrices. In this text, however, the term **scalar** is used to refer to single data items (1 x 1 matrices, as far as MATLAB is concerned). The term **vector** is used in this text to refer to single-row and single-column matrices. The term **matrix** is used to refer only to matrices that contain multiple rows and columns. This chapter explains how to use

scalars, expressions, and variables in MATLAB, reviewing some of the main points covered in Chapter 1. It describes complex number manipulation and how to solve simple equations. It also describes, in more detail than did Chapter 1, the creation and manipulation of vectors and explains the use of some of MATLAB's many vector commands. Lastly, this chapter covers the creation and manipulation of matrices. It also delves more deeply into the creation and application of X-Y plots. However, we leave a detailed discussion of plotting to Chapter 5.

Remember, it is assumed throughout this and all chapters that you are following along, using MATLAB and entering all commands shown. The answers to the Sub-Section Self-Tests are at the end of the chapter.

SCALARS

Scalar operations are those operations performed by functions that take scalars as their arguments. Of course, MATLAB views all arguments as matrices; however, many functions will not take a scalar (a matrix containing a single element) as an argument. In this section, we look at a few of the functions that do take scalars as arguments in order to see how MATLAB deals with scalars.

Numbers and Variables (`ans`, `sqrt`, `exp`)

When you enter numbers in MATLAB, you use conventional decimal notation, with an optional decimal point and leading plus or minus sign. Scientific notation uses the letter **e** to specify a power-of-ten scale factor. Imaginary numbers use either an **i** or **j** as a suffix (your choice; however MATLAB uses **i** when returning results). Some examples of legal MATLAB numbers are:

```
25
-16
3.7e-28
-0.9e41
1i
3 - 2j
```

Before starting this tutorial session, enter the `clear` command to ensure that all previous data and definitions are removed from the MATLAB workspace. Enter:

```
» clear
```

An **expression** (for example, $x+4/5$) is a mathematical construct that has a value or set of values. Like most programming languages, MATLAB uses expressions; however, unlike other programming languages, these expressions always involve entire matrices (although a matrix can be a 1 x 1 matrix). As shown in Chapter 1, the value of an expression can be found in MATLAB simply by typing the expression and pressing the **Enter** key. Enter the following to compute the value of the expression $(4.5 + 2.1) / (-2.1)^2$:

```

» (4.5+2.1)/(-2.1^2)
ans =
-1.4966

```

By default, MATLAB displays real numbers to four decimal places. Of course, you can assign a variable name to an expression (for example, $a = 4/5$). If you do not specify a variable name, MATLAB will store the result in the default variable, *ans*, as shown. Comments, which are useful in MATLAB programs, can be appended to a command line by preceding them with a percent (%) symbol.

MATLAB variable names consist of a letter, followed by any number of letters, digits, or underscores. MATLAB, however, uses only the first 31 characters of a variable name.

MATLAB is case sensitive. Users must remember this when using variables and function names. For example, MATLAB views x and X as two distinct variables. MATLAB's built-in functions use all lowercase characters in their names; the command `sqrt(10)` returns the square root of 10, but `SQRT(10)` and `Sqrt(10)` will return error messages. Enter the following:

```

» sqrt(2) % sqrt is a built-in MATLAB function
ans =
1.4142
» SQRT(2)
??? Undefined variable or capitalized internal
function SQRT; Caps Lock may be on.

```

Note that the `sqrt` function returns only the positive square root.

If an expression contains variables, the value of the expression can be computed only if the values of the variables have been previously given. For example, if you want MATLAB to calculate the value of $y = x + 5$, you must first supply a value for x , as follows:

```

» x=2
x=
2
» y=x+5 % a linear function
y=
7

```

If you have not supplied a value for x , MATLAB will return an error message.

```

» y=z+5
??? Undefined function or variable 'z'.

```

Remember from Chapter 1 that a semicolon at the end of an entry prevents MATLAB from instantly responding. Omitting the semicolon frees MATLAB to respond to the entry. Enter the following to compute $16e^{-5t}$ when $t = 5$:

```

» t=5;
» 16*exp(-5*t)
ans=
  2.2221e-010

```

MATLAB automatically displays very small and very large numbers in exponential format. The result above was 2.2221×10^{-10} . The `exp` function is one of MATLAB's "elementary functions," which include the trigonometric, exponential, complex, and rounding functions. To get a (long) list and brief description of MATLAB's elementary functions, enter the command `help elfun`.

Most of MATLAB's functions are **built-in functions**. For example, the `sin`, `cos`, and `exp` are built-in functions. However, some complex functions are implemented in **M-files**, which are ASCII text files that contain MATLAB code that implements a MATLAB program. If you write your own custom functions, the files you create will have the extension `.m` and will be called M-files. The writing of MATLAB programs is covered in Chapter 3.

When using MATLAB, you must pay close attention to the use of semicolons. A semicolon at the end of a MATLAB command suppresses the listing of the results of the command's execution. It's not uncommon for a user to forget to add the semicolon before pressing the **Enter** key when creating a large matrix or inputting a large data file and then have MATLAB start listing thousands of element values. If this should happen to you, press **Ctrl+c** to stop the listing.

Example 2-1. Numbers and Variables

Find the value of the expression $32e^{-5t}$ at $t = -3.2$.

Solution

As you solve the example problems of this chapter, remember to use the `clear` command as needed to remove all past variables from the workspace before starting each new problem. As appropriate, make sure you end commands with a semicolon to tell MATLAB not to report intermediate results. First, define the value of t .

```

» clear
» t=-3.2;

```

Then compute the value of the expression, omitting the semicolon at the end.

```

» 32*exp(-5*t)
ans =
  2.8436e+008

```

Sub-Section Self-Test 2-1

Declare the following variables, assigning the indicated values:

```
a = 1.12
b = 2.34
c = 0.72
d = 0.81
v = 19.83
```

Use the preceding variables to determine the value of the following expressions:

- a. $1 + b/v + c/v^2$
- b. $(b - a)/(d - c)$
- c. $1/(1/a + 1/b + 1/c + 1/d)$

Operator Precedence

The precedence of arithmetic operations within an expression is the same in MATLAB as in most other programming languages. The order of precedence is as follows:

1. Parentheses
2. Power (^), left to right
3. Unary plus (+), unary minus (-)
4. Multiplication (*) and division (/), left to right
5. Addition (+) and subtraction (-), left to right

When using MATLAB, you must keep the default precedence of arithmetic operations in mind. The simple problem of computing the average of two numbers demonstrates the problem.

```
» age1=30;
» age2=50;
» average=age1+age2/2 % incorrectly computes average
average =
    55
» average=(age1+age2)/2 % correctly computes average
average =
    40
```

Clearly, the second answer is the correct one. If addition had precedence over (if it were performed before) division, both calculations would have resulted in the answer 40. Of course, it is best to use parentheses liberally to ensure that MATLAB performs your computations correctly. Also, it is better to break a complex problem up into separate, easily understood, steps, rather than try to type the entire problem on one line.

Example 2-2. Precedence

A transfer function is a function that yields the output of a system, given the input to that system. Find the value of the following transfer function when $s = 0.23$.

$$H(s) = 0.3774/(s^2 + 1.0235s - 0.8877)$$

Solution

```

> s=0.23;
> Hs = 0.3774/(s^2 + 1.0235*s-0.8877)

Hs =
-0.6296

```

Sub-Section Self-Test 2-2

Find the value of the following transfer function when $s = 1.04$.

$$H(s) = 0.3774s - 3.01/(s^2 + 1.0235s + 0.912)$$

eps, Inf, NaN, and pi (realmax, realmin)

Round-off error is a concern in numerical computing. MATLAB works with limited precision due to the fact that all computers are limited in accuracy. (The smallest storage unit, one bit, must represent a finite value, however small.) MATLAB allocates eight bytes, or 64 bits, to each number it stores. The result is that MATLAB truncates all decimal expansions at the sixteenth place. Even if this is acceptable for any single calculation, its effects can accumulate over multiple iterations with unacceptable consequences.

MATLAB denotes its round off value, that is, the smallest distinguishable difference between two numbers, as **eps**. You can check the value of **eps**.

```

> eps           % the smallest difference
ans =
2.2204e-016

```

Another special value in MATLAB is **Inf**, or infinity. For example, if you divide a number by zero, you will get an error message and the value **Inf**. In the following example, x retains the value of infinity.

```

> x=1/0         % infinity
Warning: Divide by zero.
x =
Inf

```

Then there are numbers that are not numbers at all. For this situation, MATLAB assigns the value **NaN** (for “not a number”). Dividing 0 by 0 results in a value of **NaN**.

```
» x=0/0      % not a number
Warning: Divide by zero.
x =
NaN
```

Generally much more useful than **eps**, **Inf**, and **NaN** is **pi**. This built-in MATLAB constant has the value of π to the accuracy possible using 64 bits. You can display the value of **pi**, and the values of all numerical results, in either short or long format. To see how this works, enter:

```
» format short % round output to 5 digits
» pi
ans =
    3.1416
» format long % round output to 15 digits
» pi
ans =
    3.14159265358979
```

The constant **pi** is particularly useful when dealing with cyclical functions, such as those that appear in mechanical vibration and electronic signal problems. Enter:

```
» t=0.018;
» V=5*cos(2*pi*60*t)
V =
    4.3815
```

Two other values of interest are the largest and smallest real numbers MATLAB can store in the 64 bits it allocates to each number. The `realmax` and `realmin` commands give you these values. Enter:

```
» format short
» realmax      % the largest real number
ans =
    1.7977e+308
» realmin     % the smallest real number
ans =
    2.2251e-308
```

Example 2-3. eps, Inf, NaN, and pi

Determine the value of infinity divided by infinity in MATLAB.

Solution

```

> y=Inf/Inf
y =
NaN

```

Sub-Section Self-Test 2-3

Determine the values of zero divided by NaN and NaN divided by zero.

Complex Numbers (real, imag, conj, abs, angle)

Many engineering problems involve finding the roots of polynomial functions. Such functions often arise when designing or analyzing oscillating systems, such as electronic circuits and mechanical structures that are submitted to shocks. The roots of these polynomials are often complex. Therefore, all MATLAB operations and functions are designed to easily handle complex numbers. You may use either **i** or **j** to indicate the imaginary part of a complex number. (MATLAB uses **i** in the results it returns.) For example, you can define a variable *a* as complex:

```

> a=1+2*i

```

When a computed result is complex, MATLAB generates output like the following.

```

> sqrt(-43) % an imaginary number
ans=
0+6.5574i

```

The following functions are built-in MATLAB functions that work specifically with complex numbers. The arguments can be scalars, vectors, or matrices.

real	returns the real parts of the elements of a matrix
imag	returns the imaginary parts of the elements of a matrix
conj	returns the complex conjugates of the elements of a matrix
abs	returns the magnitudes of the elements of a matrix
angle	returns the phase angles, in radians, of the elements of a matrix

The following demonstrates their use. Enter:

```

> v=3+4i;
> r=real(v), i=imag(v), c=conj(v), m=abs(v), a=angle(v)
r =
3
i =

```

```

4
c =
3.0000 - 4.0000i
m =
5
a =
0.9273

```

Example 2-4. Complex Numbers

Determine the roots of the quadratic polynomial $y(x)=4x^2-2x+1$.

Solution

We will use the well-known quadratic formula, Equation (1), to solve this problem.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (1)$$

We must first define the quadratic coefficients using the form $y(x)=ax^2+bx+c$.

```
» a=4; b= -2; c=1;
```

Then we use the quadratic formula to determine the two roots.

```

» R1= (-b+sqrt (b^2-4*a*c)) / (2*a)
R1=
0.2500+0.4330i
» R2= (-b-sqrt (b^2-4*a*c)) / (2*a)
R2=
0.2500-0.4330i

```

Since our focus in this example was on simply dealing with complex numbers, we used the quadratic formula to find the roots of the equation. In Chapter 4, you will be shown how to more easily determine the roots of polynomials of any degree, including quadratic polynomials, using the MATLAB `roots` function.

Sub-Section Self-Test 2-4

Use the quadratic formula to determine the roots of the quadratic polynomial $y(x)=8x^2-x+2$.

A Simple Linear Plot (plot, grid, title, xlabel, ylabel, whos)

Creating a plot (graph) of a linear function could barely be simpler. First, you create a 2-element vector that contains the beginning and ending values of the independent variable. Then you define the function. Then you are ready to plot the function. Enter the following (of course, you can omit the comments):

```

» x=[1,8]; % a 2-element vector
» y=5.2*x-3; % another 2-element vector
» plot(x,y) % creates a linear plot
» grid % puts a grid on the plot
» title('A Linear Graph')
» xlabel('x');ylabel('5.2x-3')

```

The `grid` command places a grid on the plot, while the `title`, `xlabel`, and `ylabel` commands annotate the plot accordingly. You could have replaced the first command with `x=[1:8]` or `x=1:8` (the brackets are optional when using the colon to create a vector); however, this would have created a vector containing eight elements with the values 1, 2, 3, ... 8. Since this is a linear function, only two data points are required to create the plot.

The following code checks the size of `x` and `y` (1 x 2 matrices) and shows that their contents represent the two end points of the plot, (1,2.2) and (8,38.6). Enter:

```

» whos
  Name      Size      Bytes  Class
  x         1x2         16  double array
  y         1x2         16  double array
Grand total is 4 elements using 32 bytes
» x
x =
   1   8
» y
y =
 2.2000   38.6000

```

Chapter 5 goes into more detail on creating linear plots, as well as non-linear, polar, and 3-D plots.

Example 2-5. A Simple Linear Plot

Create a plot of the linear function $y = -2x + 2$ from $x = -3$ to $+3$.

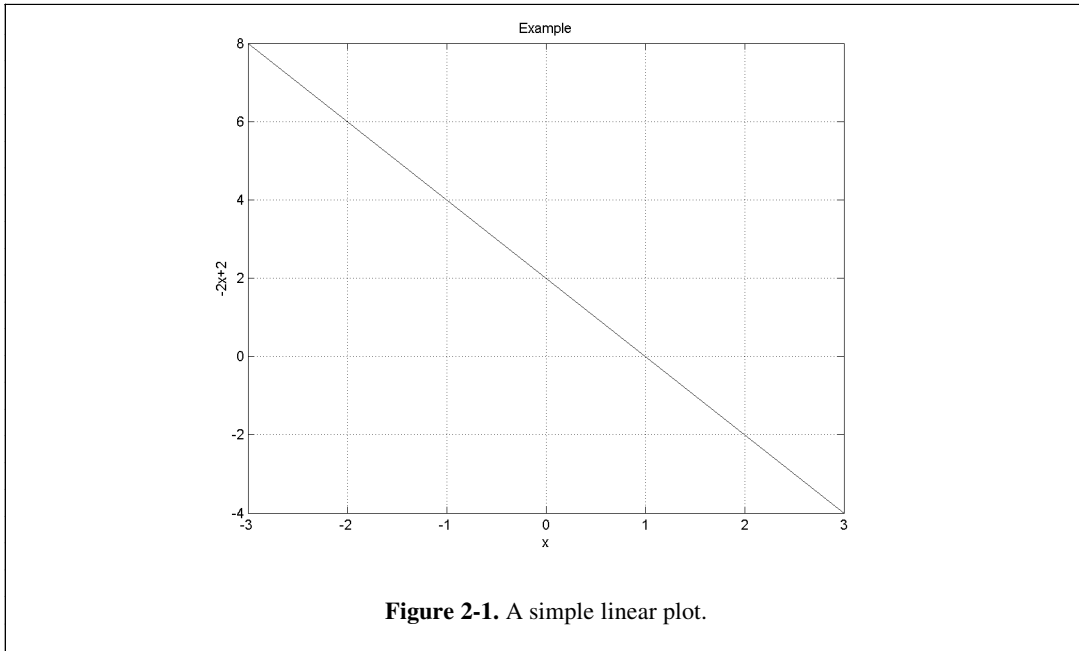
Solution

```

» x=[-3, 3]; % the x limits
» y=-2*x+2; % the y limits
» plot(x,y) % a semicolon after plot has no effect
» grid % a semicolon after grid has no effect
» title('Example');xlabel('x');ylabel('-2x+2')

```

Figure 2-1 displays the results.

**Sub-Section Self-Test 2-5**

Create a plot of the linear function $y = -2.1x - 7.2$ from $x = -3$ to $+3$.

VECTORS

In mathematics, a **vector** is a quantity, such as velocity, that is completely specified by a magnitude and a direction; an element of a vector space; or a one-dimensional array. The third meaning, a one-dimensional array, is the one assumed in this text.

Creating Vectors (linspace, logspace)

Vectors can be either row vectors or column vectors. When creating a row vector, you separate the vector elements with either spaces or commas. When creating a column vector, you separate the elements with semicolons. The following examples, respectively, create a row matrix and a column matrix. Clearing the workspace is always a good idea before starting a new problem. Enter:

```

» clear
» rowvector=[1, 2, 3, 4]    % a row vector
rowvector =
    1     2     3     4
» colvector=[1; 2; 3; 4]   % a column vector

```

```

colvector =
  1
  2
  3
  4
» whos                % check the vector sizes
Name                Size                Bytes   Class
colvector           4x1                32     double array
rowvector           1x4                32     double array
Grand total is 8 elements using 64 bytes

```

The first element in a MATLAB vector has the index 1 (not 0).

```

» rowvec=[1 2 3 4]; % this time, no commas
» rowvec(1) % the first element in 'vector'
ans =
  1
» rowvec(1)=-1; % change value of first element
» rowvec
rowvec =
 -1    2    3    4

```

As shown earlier, row vectors can be created by specifying the beginning and ending values, separated by a colon; an optional increment value can be inserted between the starting and ending values. (A value of 1 is assumed as the increment value, if it is omitted.) For example, the following command creates a vector that ranges from 1 to 1.8 in increments of 0.2. Enter:

```

» vector=[1:0.2:1.8]
vector =
  1.0000    1.2000    1.4000    1.6000    1.8000

```

The increment value can be negative. In the first of the following two examples, the increment is -2 ; in the second example, it is $-\pi/2$. Enter:

```

» values=[8:-2:0], times=[2*pi:-pi/2:0]
values =
  8    6    4    2    0
times =
  6.2832    4.7124    3.1416    1.5708    0

```

Of course, element values can be complex. Values can also be expressions, even expressions that include functions, and even vectors and matrices. (Later in this chapter, we will form matrices of vectors and matrices, a process called concatenation.) Indices too can be expressed as variables or functions, as long as the result is an integer greater than or equal to 1. Enter:

```

» x=0.5*i; c=2;
» mydata=[2*x-c cos(pi/4) exp(-1)]
mydata =
-2.0000+1.0000i      0.7071      0.3679
» mydata(c) % c=2, so we get element # 2
ans =
0.7071

```

When you want to generate linearly spaced vectors, you can use the `linspace` function to let MATLAB compute the increment for you. The `linspace` function requires three arguments: the starting value, the ending value, and the number of desired points. The two following commands are equivalent.

```

» M=0:2:10
M =
0     2     4     6     8    10
» M=linspace(0,10,6)
M =
0     2     4     6     8    10

```

In the first command, the user provides the starting and ending values and the increment value. In the second command, the user provides the starting and ending values and the number of desired elements.

Enter the following code to use MATLAB's `linspace` and `exp` functions to create a vector that contains 4 elements, evenly spaced between $-e^1$ and e^1 :

```

» vect=linspace(-exp(1),exp(1),4)
vect =
-2.7183     -0.9061     0.9061     2.7183

```

MATLAB's `logspace` function produces a logarithmically spaced vector. For example, `logspace(-2,3,100)` generates a row vector of 100 logarithmically equally spaced points between decades 10^{-2} and 10^3 . The `logspace` command is useful when plotting functions that produce outputs that fall over a large range of values and you want to use the `semilogx` function, instead of the `plot` function, to plot the values using a logarithmic X-axis. For example, enter:

```

» t=logspace(-1,1,50);
» semilogx(t,5*exp(-0.5*t));
» xlabel('t (time)'); ylabel('5exp(-0.5t)')

```

This produces the plot displayed in Figure 2-2, which has a logarithmic X-axis, along which the t values are evenly spaced. Had the `linspace` function been used, the t values would have been awkwardly (linearly) spaced along the X-axis.

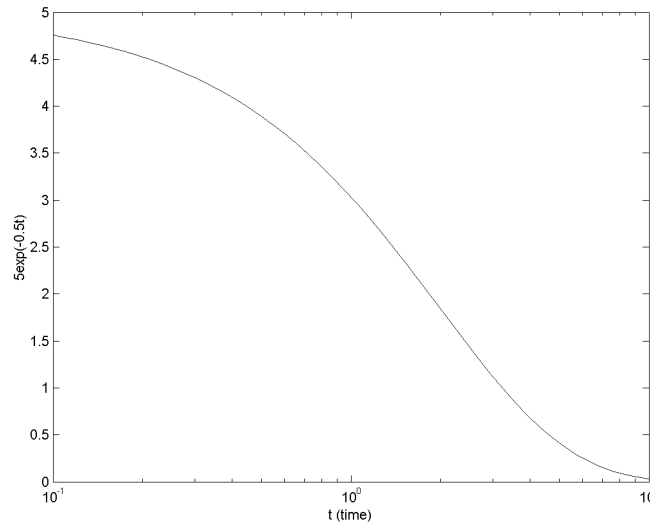


Figure 2-2. A simple logarithmic plot.

When the values are logarithmically spaced, the values are bunched toward the bottom of the range (in this case, toward the value 10^{-1}). This gives logarithmic graphing its advantage; that is, it examines more of the small values. For example, when plotting the frequency response of an audio speaker from 0 Hz to 10 kHz, you would want to use a logarithmic X-axis. Doing so will plot more of the lower values (say, from 0 to 100 Hz), giving you more information in this range. The higher values, say from 19 kHz to 10kHz will be compressed, giving you less information in this range.

You can extract small vectors from larger ones, a process called **extraction**. You can combine smaller vectors or matrices to produce larger ones, a process called **concatenation**. To extract a sub-vector, you specify the vector name and the beginning and ending indices, separated by a colon, of the elements you want to extract. For example, `vnew=vold(5:10)` creates the vector **Vnew**, which contains elements 5 through 10 from the vector **Vold**.

To concatenate two or more vectors or matrices side-by-side, as a row vector, merely list their names within brackets, separated by spaces or commas. To concatenate them one above the other, as a column vector, list their names within brackets, separated by semicolons.

From this point forward in the text, it is assumed that you will enter and execute the code following the `»` prompt. Extraction and concatenation are both demonstrated in the following code.

```

» V1=[1, 3, 5]; % a 3-element row vector
» V2=[9, -1, 11]; % a 3-element row vector
» V3=V1(2:3) % extract elements 2 to 3 of V1
V3 =
     3     5
» V4=[V1, V2] % concatenate V1, V2 side-by-side
V4 =
     1     3     5     9    -1    11

```

```

      1      3      5      9     -1     11
» V5=[V1; V2] % concatenate V1, V2 one above the other
V5 =
      1      3      5
      9     -1     11

```

Extraction works in the same way for both row vectors and column vectors. For example, the command `V3=V1(2:3)` creates a vector **V3** that consists of the second and third element of vector **V1**. Whether **V1** is a row or column vector, **V3** will be the same type. Concatenating column vectors works as follows:

```

» V1=[5; 6]; % a 2-element column vector
» V2=[6; 7]; % another 2-element column vector
» V3=[V1; V2] % a 4-element column vector
V3 =
      5
      6
      6
      7

```

Another way to create a vector is to extract it from an existing matrix, by deleting all but one row or column in the matrix. You use a colon to designate either all rows or all columns. In the following example, `A(:,3)` refers to all rows of column 3 of matrix **A**, and `A(1,:)` refers to all columns of row 1 of matrix **A**.

```

» A=[1,2,3,4; 5,6,7,8; 2,4,6,8] % a 3 x 4 matrix
A =
      1      2      3      4
      5      6      7      8
      2      4      6      8

» B=A(:,3) % extract all rows of column 3
B=
      3
      7
      6

» C=A(1,:) % extract all columns of row 1
C=
      1      2      3      4

```

You do not have to extract all rows in a column or all columns in a row. For example, the following sets **N** equal to columns 10 through 20 in row 5 of matrix **M**.

```

» N = M(5,10:20);

```

A later section in this chapter shows you more about how to create matrices and goes more deeply into extracting vectors from matrices. That section also shows you how to combine vectors to form matrices.

Example 2-6. Creating Vectors

Create two vectors, the first containing five linearly spaced values from 1 to 10, the second containing five logarithmically spaced values from 1 to 10. In what way do the two vectors' element values differ?

Solution

We use the `linspace` function to create the linearly spaced vector. In the argument list, we specify the starting value, the ending value, and the number of values desired.

```
> linear=linspace(1,10,5)
linear =
    1.0000    3.2500    5.5000    7.7500   10.0000
```

When using the `logspace` function, the first and second arguments are powers of 10, indicating the starting and ending values. To produce values from 1 to 10, these arguments must be 0 and 1, giving us values that range from 10^0 to 10^1 . The third argument is the number of values desired.

```
> log=logspace(0,1,5)
log =
    1.0000    1.7783    3.1623    5.6234   10.0000
```

When using `logspace`, more low values and fewer high values are produced than when using the `linspace` function.

Sub-Section Self-Test 2-6

Create two row vectors, **V1** with integer values from 1 to 100 and **V2** with integer values from 101 to 200. Then, using a single command, create a row vector **V3** that contains the extracted last five elements of **V1** and the extracted first five elements of **V2**. **V3** will contain the values 96, 97, ... 105.

Basic Vector Functions (size, length, min, max, sum, sort, abs)

MATLAB includes a number of built-in functions that you can use to determine a number of characteristics of a vector. The following are the most commonly used such functions.

<code>size</code>	Returns the dimensions of a matrix
<code>length</code>	Returns the number of elements in a matrix
<code>min</code>	Returns the minimum value contained in a matrix
<code>max</code>	Returns the maximum value contained in a matrix
<code>sum</code>	Returns the sum of the elements in a matrix

sort	Returns the sorted elements in a matrix
abs	Returns the absolute values of the elements in a matrix

The following code demonstrates the use of these functions. The first function used, `size`, indicates that **V1** is a 1 x 3 matrix; that is, **V1** has one row and three columns.

```

» V1=[-5, -3, -1];
» V2=[3, 1, 2, 4];
» numvals=size(V1)
numvals =
1     3
» len=length(V1)
len =
3
» smallest=min(V2)
smallest =
1
» largest=max(V2)
largest =
4
» total=sum(V2)
total =
10
» V3=sort(V2)
V3 =
1     2     3     4
» V4=abs(V1)
V4 =
5     3     1

```

Example 2-7. Basic Vector Functions

Create a column vector **Colvec** that contains the values from -50 to +50. Then use MATLAB's built-in functions to determine the number of elements in **Colvec**, the sum of elements in **Colvec**, and the sum of the absolute values of the elements in **Colvec**.

Solution

```

» Colvec=-50:50;
» length(Colvec)
ans =
101
» sum(Colvec)
ans =
0

```

```

» sum(abs(Colvec))
ans =
    2550

```

Sub-Section Self-Test 2-7

MATLAB includes a number of functions that perform statistical analysis on vectors (and matrices). Notice that the above list of vector functions does not contain a function that computes the average of a vector's element values. Enter the command `lookfor average` to find the name of the function that performs the average operation.

Vector Arithmetic - Array Operations (+, -, .*, ./, .^, abs)

When we move from scalars to vectors (and matrices), some confusion arises when performing arithmetic operations because we can perform some operations either on an element-by-element basis or on the vectors or matrices as whole entities. MATLAB calls the first type of operation an **array operation** and calls the second type a **matrix operation** (the topic of the next sub-section). Addition, subtraction, and `abs` (computes the absolute value on an element-by-element basis) are examples of array operations. The matrix inverse, determinant, and dot product are examples of matrix operations.

Multiplication, division, and exponentiation (power) are examples of operations that can be either an array operation or a matrix operation. In cases such as these, the operator (`*`, `/`, `^`) is preceded by a period when an array operation is desired. If a matrix operation is desired, the period is omitted. As the following code shows, squaring each individual element in a vector is an array operation.

```

» A=[1 2 3 4 5]; % omitted the commas this time
» B=A.^2 % must use the period!
B =
    1     4     9    16    25
» C=sqrt(B)
C =
    1     2     3     4     5

```

Notice that `sqrt` computed the square root of **B** on an element-by-element basis, making `sqrt` an array operation. Had you entered `B=A^2`, MATLAB would have attempted to compute the square of **A**. However, in order to have a square (or any other power value), a matrix must be a square matrix, having the same number of rows as it has columns.

All array operations involving multiple vectors (or matrices) require that those vectors (or matrices) be of equal size. The following demonstrates addition, subtraction, multiplication, and division as array operations.

```

» A=[1, 2, 3, 4, 5];
» B=[5, 4, 3, 2, 1];
» C=A+B, D=A-B, E=A.*B, F=A./B
C =
     6     6     6     6     6
D =
    -4    -2     0     2     4
E =
     5     8     9     8     5
F =
    0.2000    0.5000    1.0000    2.0000    5.0000

```

Had you forgotten to precede the * and / symbols with periods (which MATLAB users often do) in the above entries, MATLAB would have attempted matrix operations and the following would have occurred.

```

» E=A*B
??? Error using ==> *
Inner matrix dimensions must agree.
» F=A/B
F =
    0.6364

```

In the first case, an error occurred. In order to multiply **A** by **B**, **A** must have the same number of columns as **B** has rows. In the second case, the result may not be what the user expected. Multiplication and division as matrix operations are covered later in this chapter.

Example 2-8. Vector Arithmetic - Array Operations

Create a vector **X** that contains the numbers -1.0, -0.5, 0, +0.5, and +1.0. Then create a vector **Y** that contains the numbers $Y = X^3 - X$ on an element-by-element basis.

Solution

```

» X=-1:0.5:1
X =
    -1.0000    -0.5000     0     0.5000     1.0000
» Y=X.^3-X
Y =
    0.3750     0    -0.3750     0

```

Sub-Section Self-Test 2-8

Create a vector **X** that contains the numbers -1, 1, 2, 4, and 5. Then create a vector **Y** that contains the numbers $Y = X^2 + 3X - 1$ on an element-by-element basis.

A Simple Vector Plot (plot, title, xlabel, ylabel)

The `plot(x,y)` function plots vector **x** against vector **y**. The two vectors must have the same length, as each point (x,y) in the plot consists of corresponding paired elements in **x** and **y**. The two vectors **x** and **y** can be both row vectors, both column vectors, or one row vector and one column vector. The `plot(y)` function plots the elements in the vector **y** against their indices. When any of the elements in **y** are complex, then `plot(y)` is equivalent to `plot(real(y), imag(y))`; that is, the real parts of the elements are plotted against the imaginary parts.

As shown previously, the `plot` function is typically augmented with the `title` function, which adds a title at the top of a plot; the `xlabel` and `ylabel` functions, which label the axes; and the `grid` function, which adds a grid to the plot to make it easier to read. As described in Chapter 5, MATLAB includes a number of other functions that one can use to annotate plots still further. The `plot` function includes a number of options, also described in Chapter 5, that can be used to improve the appearance and information content of plots. For example, you can set the line color to red and the line type to dashed with the command `plot(c,y,'r:')`. By default, MATLAB plots solid blue lines.

Example 2-9. A Simple Vector Plot

The following data represent voltages measured at times given in seconds. Plot the voltages against the times, using a black dashed line. The letter 'k' is used to indicate black.

```
Time (sec.) = 0  2  3  6  8  9  12  18
Voltage (volts) = 1.21  1.42  1.22  1.53  1.12  1.33  1.01  1.51
```

Solution

```
> time=[0,2,3,6,8,9,12,18];
> volts=[1.21,1.42,1.22,1.53,1.12,1.33,1.01,1.51];
> plot(time,volts,'k:');
> title('Voltage Plot');
> xlabel('Time (seconds)');ylabel('Voltage (volts)')
```

Figure 2-3 at the top of the next page displays the results.

Sub-Section Self-Test 2-9

The following data represent the deflection of a cantilevered beam, measured in millimeters, measured by a strain gauge. The measurements are taken each millisecond for 15 milliseconds. Use the `plot(y)` form of the `plot` function to plot these data using a solid red line.

```
Deflection = 1.1, 1.3, 0.1, -0.1, -1.1, -0.5, 0.0, 0.2, 0.4, 0.1, -0.2, -0.8, -0.1, 0.2, 0.3
```

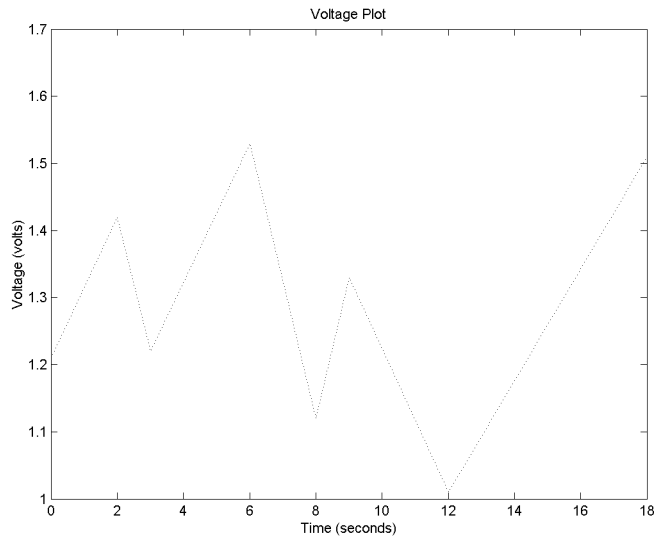


Figure 2-3. A simple vector plot.

MATRICES

While MATLAB treats scalars as 1×1 matrices, row vectors as $1 \times n$ matrices, and column vectors as $n \times 1$ matrices, in this text, only 2-dimensional arrays are called matrices.

Creating Matrices

As with vectors, you can define matrices explicitly, or create them by combining elements from other matrices, or by concatenating other matrices. When defining matrices explicitly, you use either spaces or commas (your choice) to separate elements in a row. You use semicolons to separate rows. Earlier in this chapter, we noted that you separate elements in a row vector with either spaces or commas. This is because MATLAB views each element in a row vector as a column. When defining a column vector, you separate the elements, each seen as a row, with semicolons.

The following example creates a 2×4 matrix. As usual, you should clear the workspace before starting a new problem.

```

» clear                                % clear the workspace
» M=[1 2 3 4; 5 6 7 8] % or M=[1,2,3,4; 5,6,7,8]
M =
     1     2     3     4
     5     6     7     8
» whos                                 % check the size of M
Name   Size  Bytes   Class
M      2x4   64      double array
Grand total is 8 elements using 64 bytes

```

Individual matrix elements are specified with indices within parentheses, as in $M(1,1)$. The first number in parentheses specifies the row; the second specifies the column. Row and column indices begin at 1; therefore, $M(1,1)$ is the element in the upper left-hand corner of the matrix \mathbf{M} . The following code creates the matrix \mathbf{M} and then examines the value of the upper left element and the value of the lower right element.

```

» M=[1 2 3 4; 5 6 7 8];
» M(1,1),M(2,4)
ans =
    1
ans =
    8

```

If \mathbf{M} is an $m \times n$ matrix, then referencing $M(i,j)$, where i is less than 1 or greater than m , or j is less than 1 or greater than n , yields an error message. For example, if \mathbf{M} is a 2 x 4 matrix, $M(4,4)$ does not exist, and the following occurs:

```

» M(4,4)
??? Index exceeds matrix dimensions.

```

In formal mathematical notation, an uppercase letter usually designates matrix names, whereas individual matrix elements are designated by lowercase letters. However, since MATLAB is case sensitive, you must consistently use the same case whether designating a complete matrix or just one of its elements. The following MATLAB session demonstrates the principle:

```

» M=[0 1 2 3; 4 5 6 7; 8 9 0 1]
M=
    0     1     2     3
    4     5     6     7
    8     9     0     1
» M(2,3)
ans=
    6
» m(2,3)
??? Undefined function or variable 'm'.

```

Earlier in the chapter, we saw how to create large vectors that contain evenly spaced element values. For example, the commands $X=0:99$ and $Y=0:0.1:9.9$ would each create a 100-element row vectors. You can use the same technique to create large matrices:

```

» M=[1:5; -1:-1:-5; -3:1; 1.1:0.1:1.5]
M =
    1.0000     2.0000     3.0000     4.0000     5.0000
   -1.0000    -2.0000    -3.0000    -4.0000    -5.0000
    -3.0000     -2.0000     -1.0000     0.0000     1.0000
    1.1000     1.2000     1.3000     1.4000     1.5000

```

```

-3.0000    -2.0000    -1.0000         0     1.0000
 1.1000     1.2000     1.3000     1.4000     1.5000

```

As stated in the above section on vectors, elements can be expressions, even expressions that use functions. For example, a 2 x 2 matrix can be declared as follows:

```

» A=[4/5, sqrt(2); exp(-1), cos(pi/8)]
A=
 0.8000         1.4142
 0.3678         0.9239

```

Complex matrices can be defined in the same way. A 2 x 1 complex matrix can be defined so that

```

» M=[(1+1i)*(1-2i); 2-3*(0-2i)]
M =
 3.0000 - 1.0000i
 2.0000 + 6.0000i

```

Note that a 1 must precede the i (or j), as in 1i; otherwise MATLAB will interpret the i as a variable name.

You can even use the `linspace` and `logspace` functions to create a matrix:

```

» M=[linspace(1,5,3); logspace(1,5,3)]
M =
 1         3         5
 10      1000 100000

```

You can create matrices by combining vectors. For example, if **a**, **b**, and **c** are all row vectors of the same length, they can be combined to form a 3-row matrix **D** as follows:

```

» a=[1, 2, 3]; b=[4, 5, 6]; c=[7, 8, 9];
» D=[a; b; c]
D=
 1     2     3
 4     5     6
 7     8     9

```

Note that, in the definition of **D**, semicolons separate the vector names. As when creating any matrix, semicolons delimit rows; therefore, **a** is the first row, **b** is the second row, and **c** is the third row.

The following is another example of concatenation.

```

» x=[1, 2, 3, 4];
» y=x+1;
» z=[x; y]

```

```

z =
  1   2   3   4
  2   3   4   5

```

Here, the matrix **x** became the first row of the matrix **z**, while the matrix **y** became the second row of the matrix **z**.

By deleting rows or columns from a matrix, you can create smaller matrices from larger ones. You use an empty pair of brackets (`[]`) to delete a matrix row or column. The following example creates a 3 x 3 matrix **M**, sets the matrix **N** equal to **M**, and then deletes the middle column of the matrix **N**. A colon is used to designate either all rows or all columns; therefore, in the following example, `N(:,2)` refers to all rows (each a single element) of column 2 of the matrix **N**.

```

>> M=[1 2 3; 3 4 5; 4 5 6] % create 3 x 3 M
M =
  1   2   3
  3   4   5
  4   5   6
>> N=M;
>> N(:,2)=[] % delete all rows in column 2
N =
  1   3
  3   5
  4   6

```

To create a matrix **P** that lacks the second row of **M**, you would enter the following commands.

```

>> P=M;
>> P(2,:)=[] % delete all columns in row 2
P =
  1   2   3
  4   5   6

```

You can extract matrices from matrices. For example, the following extracts the third and fourth elements in rows two and three of the matrix **M**.

```

>> M=[1 2 3 4; 2 3 4 5; 3 4 5 6; 4 5 6 7]
M =
  1   2   3   4
  2   3   4   5
  3   4   5   6
  4   5   6   7
>> N=M(2:3,3:4) % rows 2 and 3, columns 3 and 4

```

```
N =
    4     5
    5     6
```

You will find the ability to extract columns and rows from matrices useful when graphing data. For example, a data file may contain a 2-column matrix, in which the first column contains measurements and the second column contains times that correspond to the measurements. In order to plot measurements versus times using `MATLAB`, you would plot the first column against the second after converting each to a vector, as shown in the next example.

There are still other ways to create matrices. For example, Chapter 3 describes how to create matrices by reading data files. Certain matrix operations and functions also create matrices. Many of these operations and functions are described throughout the remainder of this text.

Example 2-10. Creating Matrices

Create a vector from the first column of matrix **X**, naming the vector **time**. Then create a vector from the second column of matrix **X**, naming the vector **stress**.

```
X =
    3.2
    4.4
    4.8
    3.8
    4.1
    3.9
```

Solution

First, define the matrix **X** (which more likely would be read from a text data file, as explained in Chapter 3). Then create the vectors **time** and **stress**.

```
» X=[1 3.2; 2 4.4; 3 4.8; 4 3.8; 5 4.1; 6 3.9]
X =
    1.0000    3.2000
    2.0000    4.4000
    3.0000    4.8000
    4.0000    3.8000
    5.0000    4.1000
    6.0000    3.9000
» time=X(:,1)      % extract all rows in column 1
time =
     1
     2
     3
     4
     5
     6
```

```

> stress=X(:,2)           % extract all rows in column 2
stress =
    3.2000
    4.4000
    4.8000
    3.8000
    4.1000
    3.9000

```

The command `plot(time, stress)` would produce a graph of stress versus time.

Sub-Section Self-Test 2-10

Create the matrix `x` shown in the preceding example. Then create a column vector `Y` in which each row element is the sum of the two elements in each corresponding row of `X`.

Basic Matrix Functions (size, length, min, max, sum, sort, abs)

Earlier in this chapter, we mentioned a set of built-in MATLAB functions that you can use to determine the basic characteristics of a vector. These same functions are applicable to matrices as well. The following are the most commonly used of these functions.

<code>size</code>	Returns the size of a matrix
<code>length</code>	Returns the number of columns in a matrix
<code>min</code>	Returns the minimum value in each column in a matrix
<code>max</code>	Returns the maximum value in each column in a matrix
<code>sum</code>	Returns the sum of the values in each column in a matrix
<code>sort</code>	Returns the sorted elements of each column in a matrix
<code>abs</code>	Returns the absolute values of the elements in a matrix

The following code demonstrates the use of these functions.

```

> M=[4 3 2 1; -4 -3 -2 -1; 1 2 3 4]
M =
     4     3     2     1
    -4    -3    -2    -1
     1     2     3     4
> size(M), small=min(M), big=max(M)
ans =
     3     4
small =
    -4    -3    -2    -1
big =
     4     3     3     4

```

```

» sum(M), sort(M), abs(M)
ans =
     1     2     3     4
ans =
    -4    -3    -2    -1
     1     2     2     1
     4     3     3     4
ans =
     4     3     2     1
     4     3     2     1
     1     2     3     4

```

Example 2-11. Basic Matrix Functions

Create a matrix **M** that contains three rows. The first row contains 5 linearly spaced values from -5 to -1. The second row contains 5 linearly spaced values from -2 to +2. The third row contains 5 linearly spaced values from 1 to 5. Then use MATLAB's built-in functions to determine the sum of the elements in **M**.

Solution

```

» M=[linspace(-5,-1,5); linspace(-2,2,5); linspace(1,5,5)]
M =
    -5    -4    -3    -2    -1
    -2    -1     0     1     2
     1     2     3     4     5
» total=sum(sum(M))
total =
     0

```

Note that the command `sum(M)` returns the vector `[-6 -3 0 3 6]`, which contains the column sums of **M**. Then `sum(sum(M))` returns the sum of -6, -3, 0, 3, and 6.

Sub-Section Self-Test 2-11

Create a matrix **M** that contains two rows. The first row contains 7 logarithmically spaced values from 0.01 to 1. The second row contains 7 linearly spaced values from 1 to 100. Then use MATLAB's built-in functions to determine the smallest and largest of the sums of the columns in **M**.

Matrix Arithmetic - Array Operations (+, -, .*, ./, .^, abs)

As stated earlier in this chapter, an **array operation** is one in which the operation is performed on an element-by-element basis within a vector or matrix. Addition and subtraction of matrices are always array operations. Squaring a matrix is an array operation only if done on an element-by-element basis. An array operation is distinguished by a period preceding the

operator, as in `.*` or `.^`. In the following example, each individual element of matrix **A** is squared and then 2 is added to each element.

```

» A=[1 2 3; 2 3 4]
A =
     1     2     3
     2     3     4
» B=(A.^2)+2
B =
     3     6    11
     6    11    18

```

The common arithmetic operations, addition, subtraction, multiplication, and division work in exactly the same way as they do with vectors. When performing these operations on two or more matrices, all of the matrices must be the same size, having the same number of rows and the same number of columns.

```

» clear % clear the workspace
» A=[1 2 3; 4 5 6], B=[3 2 1; 6 5 4]
A =
     1     2     3
     4     5     6
B =
     3     2     1
     6     5     4
» C=A+B, D=A-B, E=A.*B, F=A./B
C =
     4     4     4
    10    10    10
D =
    -2     0     2
    -2     0     2
E =
     3     4     3
    24    25    24
F =
    0.3333    1.0000    3.0000
    0.6667    1.0000    1.5000

```

Had you forgotten to precede the `*` and `/` symbols with periods in the above entries, MATLAB would have attempted matrix operations, the topic of the next sub-section.

The `abs`, `sqrt`, and other MATLAB functions perform, by default, array operations, performing their operations on an element-by-element basis.

Example 2-12. Matrix Arithmetic - Array Operations

Create a matrix **X** that contains the numbers:

```
-2  -1  0  1
-1  0  1  2
0   1  2  3
```

Then create a vector **Y** that contains the numbers

$Y = X^3 - 1$ on an element-by-element basis.

Solution

```
» clear
» X=[-2 -1 0 1; -1 0 1 2; 0 1 2 3]
X =
    -2    -1     0     1
    -1     0     1     2
     0     1     2     3
» Y=X.^3-1
Y =
    -9    -2    -1     0
    -2    -1     0     7
    -1     0     7    26
```

Note that the -1 is added to each element of the matrix after each element is cubed

Sub-Section Self-Test 2-12

Create a matrix **X** (the same matrix as in the above example) that contains the numbers:

```
-2  -1  0  1
-1  0  1  2
0   1  2  3
```

Then create a vector **Y** that contains the numbers $Y = X^2 + 3X - 1$ on an element-by-element basis.

Matrix Arithmetic - Matrix Operations (*, /, ^)

A **matrix operation** works on vectors or matrices as whole entities. Examples include multiplying or dividing two matrices and finding the inverse, determinant, or dot product of a matrix. Here, we will just introduce the concept of the matrix operation, leaving a more thorough discussion of matrix operations to Chapter 4.

In order to compute the matrix product $\mathbf{A}*\mathbf{B}$, \mathbf{A} must have the same number of columns as \mathbf{B} has rows. That is, if the dimensions of \mathbf{A} are $m \times n$, then the dimension of \mathbf{B} must be $n \times p$. The dimensions of $\mathbf{A}*\mathbf{B}$ will be $m \times p$. For example:

```

» A=[0 1 2; 1 2 3] % a 2 x 3 matrix
A =
    0     1     2
    1     2     3
» B=[0 1; 1 2; 2 3] % a 3 x 2 matrix
B =
    0     1
    1     2
    2     3
» C=A*B % should be a 2 x 2 matrix
C =
     5     8
     8    14
» D=B*A % should be a 3 x 3 matrix
D =
     1     2     3
     2     5     8
     3     8    13

```

In the above example, we multiplied a 2 x 3 matrix by a 3 x 2 matrix, and the result was a 2 x 2 matrix. We multiplied a 3 x 2 matrix by a 2 x 3 matrix, and the result was a 3 x 3 matrix.

In order to compute the power (square, cube, etc.) of a matrix, the matrix must be a square, that is, having the same number of columns as it has rows. In the following example, the matrix A is squared.

```

» A =
     1     2
     2     3
» B=A^2
B =
     5     8
     8    13

```

Note that computing the power of a square matrix is actually a normal matrix multiplication process in which the left matrix has the same number of columns as the right matrix has rows. Matrix multiplication and division are covered in more detail in Chapter 4. Chapter 4 also describes a number of other matrix operations that are relevant to engineering, including the dot and cross products and solving simultaneous equations.

Example 2-13. Matrix Arithmetic - Matrix Operations

Create a row vector **A** that contains the elements 1, 2, 3, and 4 and a column vector with the elements 1, 2, 3 and 4. Note that **A** is a 1 x 4 matrix and **B** is a 4 x 1 matrix. Create the matrices **C** = **A*B** and **D** = **B*A**. Although you cannot compute \mathbf{A}^{-1} or \mathbf{B}^{-1} (neither **A** nor **B** is square), you can divide each matrix by itself. Compute **A/A**.

Solution

```

» A=[1 2 3 4]
A =
     1     2     3     4
» B=[1;2;3;4]
B =
     1
     2
     3
     4
» C=A*B, D=B*A, A/A
C =
    30
D =
     1     2     3     4
     2     4     6     8
     3     6     9    12
     4     8    12    16
ans =
     1

```

Note that the dimensions of **C** (1 x 1) and **D** (4 x 4) are what we would expect. Likewise, we might expect **A/A** to be equal to 1, and that any matrix divided by itself would result in a value of 1. However, as we will see in Chapter 3, the inverse of some matrices (for example, $1/\mathbf{D}$) is equal to infinity, resulting in a quotient (for example, \mathbf{D}/\mathbf{D}) of infinity, rather than the expected 1.

Sub-Section Self-Test 2-13

Using **A** and **B** from the above example, compute **A*B** and **B*A**. Why do you get an error message when you attempt to compute **A/B** and **B/A**?

Introducing Text (whos)

Matrices of text elements can also be created in MATLAB. Text, or character, elements must be delimited by single quotes. Any keyboard character, including numbers and punctuation, can be contained in the quotes. Note that, as with all matrices, the rows must all be the same size; therefore, in the following example, a space was added at the end of Sam so that each row includes four elements, each a single character. The following creates a 3 x 4 matrix of character elements:

```

» clear
» names=['Jill'; 'Mary'; 'Sam ']
names =
Jill
Mary
Sam
» whos
Name          Size          Bytes  Class
names         3x4             24    char array
Grand total is 12 elements using 24 bytes

```

The result here was a 3 x 4 matrix. In MATLAB, each character consumes 2 bytes; therefore, the 12-element matrix **names** consumes 24 bytes of memory.

Example 2-14. Dealing with Text

Create a matrix that contains the names James, Dare, and Allison. Create a second matrix that contains the words first, second, and third. Remember to pad shorter terms with blanks so that the rows of each matrix all have the same number of character elements. Concatenate the two matrices so that the names form the right column and the positions form the left column. Insert extra blanks after the positions in order to separate the two columns of information.

Solution

```

» names=['James '; 'Dare  '; 'Allison'];
» position=['first  '; 'second  '; 'third   '];%spaces added
» result=[position names] %combine 2 vectors
result =
first James
second Dare
third Allison
» size(result)
ans =
    3    15

```

Note that the matrix result is a 3 x 15 matrix, not a 3 x 2 matrix because each character, not each name, is an element. Also note that here you combined two text matrices to form a larger text matrix; however, you cannot combine text and numbers in a single matrix. Text and numbers can be stored only in structure arrays (described in Chapter 6).

Sub-Section Self-Test 2-14

Create a matrix that contains the California oil reservoir names West Side, Lower Kern, Lost Hills. Create a second matrix that contains the well numbers 11-1, 13-2, and 14-1. Concatenate the two matrices so that the reservoir names form a left column and the well numbers form a right column.

A Simple Matrix Plot (plot, xlabel, ylabel, title, grid)

When you want to plot data that is in a matrix, that matrix often has a form in which either the first row or first column contains a series of reference points, and each of the remaining rows or columns contain a series of measurements taken at the corresponding reference points. For example, in the matrix below, column 1 contains time in milliseconds and each of the three non-time columns below contain voltage measurements taken at four different places in an electronic circuit.

Time	Va	Vb	Vc
0	4.89	4.86	4.99
1	4.72	4.82	4.86
2	4.68	4.70	4.73
3	4.62	4.68	4.71
4	4.58	4.60	4.63

When plotting data such as that in the above matrix, one wants a separate curve for each column, creating a plot containing multiple curves.

Data like that above is often collected by specially designed data acquisition systems (DAS) and stored in large data files before being passed on to MATLAB for plotting and/or analysis. The example below demonstrates how to create a plot that displays all of the data in the above matrix, assuming that data is in a data file. Chapter 3 describes the use of data files. Chapter 5 goes into more detail on plotting data contained in matrices.

Example 2-15. A Simple Matrix Plot

A data file called `voltdata.txt` contains a matrix of data, in which the first element (called a field) in each row (called a record) is a time in milliseconds and the remaining three elements are voltages measured at three different points a, b, and c in an electronic circuit. Read the data file and plot its contents. Graph each voltage set individually in a single plot.

Solution

First, we read the data file and check its contents.

```

» clear
» load voltdata.txt
» voltdata % check the contents of the data file
voltdata =
    0    4.8900    4.8600    4.9900
  1.0000    4.7200    4.8200    4.8600
  2.0000    4.6800    4.7000    4.7300
  3.0000    4.6200    4.6800    4.7100
  4.0000    4.5800    4.6000    4.6300

```

Now, the first column contains timing data, which we will plot along the X-axis. The data in each of the remaining three columns we now plot as three separate curves.

```

» plot(voltdata(:,1),voltdata(:,2),'ko-')
» hold on % holds all subsequent plots
» plot(voltdata(:,1),voltdata(:,3),'ks-')
» plot(voltdata(:,1),voltdata(:,4),'kd-')
» title('Circuit Voltages')
» xlabel('Time (ms)')
» ylabel('Volts: o=point a; x=point b; +=point c')

```

The `hold on` command holds plotted curves in the Figure Window while more curves are plotted. Without it, the results of each `plot` command would replace the results of the previous `plot` command. The `plot` argument `voltdata(:,1)` is a column vector containing all rows in column 1 of the matrix **voltdata**. In the third `plot` argument, 'k' tells MATLAB to draw a black curve, 'o' calls for circles at the plotted data points, 's' calls for squares, 'd' calls for diamonds, and the dash (-) tells MATLAB to draw solid lines. Chapter 5 describes many more `plot` options. For example, MATLAB can draw each curve in a different color.

Sub-Section Self-Test 2-15

Manually enter the **voltdata** matrix in the previous example and plot a single curve - the average of the three data sets.

ENGINEERING APPLICATION - ECONOMIC BREAKEVEN POINT

The cost of producing a product generally decreases with the number of units produced. Of course, revenue increases with the number of units sold. In engineering economics, the **breakeven point** is that number of units produced and sold when costs equal revenues.

Assume that analysis has shown that the cost, in dollars, of producing widgets is given by the equation $cost(w)=3000+5w$. That is, it costs \$3,005.00 to produce one widget, but it costs \$8.00 ($(3000+5000)/1000$) per widget to produce 1,000 widgets. If we sell our widgets for \$12.50 each, our revenues are defined by the equation $revenue(w)=12.50w$. Plot the two functions in order to determine the breakeven point.

To solve this problem, we first define the range of the independent variable *widgets*, the number of widgets produced. We will assume the breakeven point is less than 1000 widgets.

```
» widgets=0:1000;
```

Now, we define the two relevant functions.

```
» costs=3000+5*widgets;
» revenues=12.50*widgets;
```

And now, plot the two curves.

```
» plot(widgets, costs)
» hold on
» plot(widgets, revenues)
» grid
» xlabel('Numbers of Widgets')
» ylabel('Dollars')
```

Figure 2-4 now shows the breakeven to be 400 widgets. If we produce and sell 400 widgets, we make no profit, but neither do we lose any money.

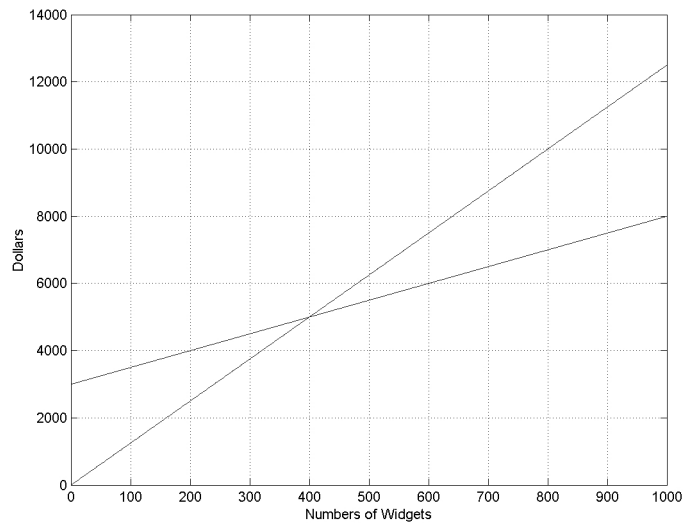


Figure 2-4. Breakeven point analysis.

CHAPTER SUMMARY

This chapter introduced many of the basic concepts of using MATLAB and described how to create simple plots. The main topics covered were:

- Scalars, numbers, and variables
- Operator precedence
- The special values `eps`, `Inf`, `NaN`, and `pi`
- Complex numbers
- Creating vectors
- Basic vector functions
- Vector arithmetic
- Creating matrices
- Basic matrix functions

- Matrix arithmetic - array operations
- Matrix arithmetic - matrix operations
- Simple text handling
- Simple plotting of linear functions, vectors, and matrices

KEY TERMS USED IN THIS CHAPTER

Array operation	Expression	M-files
Breakeven point	Matrix	Scalar
Built-in function	Matrix operation	Vector

COMMANDS AND FUNCTIONS USED IN THIS CHAPTER

For more information on any command or function, at the Command Window prompt enter `help` and then the name of the command or function.

<code>abs</code>	Returns the absolute value of x of the elements in a matrix
<code>angle</code>	Returns the phase angles, in radians, of the complex elements in a matrix
<code>conj</code>	Returns the complex conjugates of the elements in a matrix
<code>exp</code>	Returns the values e^x of the elements in a matrix
<code>grid</code>	Places a grid pattern on a plot
<code>imag</code>	Returns the imaginary parts of the complex elements in a matrix
<code>length</code>	Returns the number of elements in a vector or number of rows in a matrix
<code>linspace</code>	Returns a vector that contains equally spaced elements from a lower limit to an upper limit
<code>loglog</code>	Produces a log-log X-Y plot with logarithmic X- and Y-axes
<code>logspace</code>	Returns a vector that contains logarithmically spaced elements from a lower limit to an upper limit
<code>max</code>	Returns the maximum value in a vector or in each column in a matrix
<code>min</code>	Returns the minimum value in a vector or in each column in a matrix
<code>plot</code>	Produces an X-Y plot (graph)
<code>real</code>	Returns the real parts of the complex elements in a matrix
<code>realmax</code>	Returns the largest floating point number the computer can represent
<code>realmin</code>	Returns the smallest floating point number the computer can represent
<code>semilogx</code>	Produces a semilog X-Y plot with a logarithmic X-axis
<code>semilogy</code>	Produces a semilog X-Y plot with a logarithmic Y-axis
<code>size</code>	Returns the size of a matrix
<code>sort</code>	Returns the sorted columns in a matrix
<code>sqrt</code>	Returns the positive square roots of the elements in a matrix
<code>sum</code>	Returns the sum of a vector or of each column in a matrix
<code>title</code>	Puts a title on the current plot
<code>xlabel</code>	Puts a label on the X-axis of the current plot
<code>ylabel</code>	Puts a label on the Y-axis of the current plot.

SPECIAL CHARACTERS USED IN THIS CHAPTER

()	Contain the index of a matrix element or the arguments of a function
'	Delimits text in a character matrix
,	Separates elements of a matrix row or matrix subscripts
:	Delimits starting and ending values in a range of values and creates a vector from a matrix
;	Suppresses MATLAB responses and separates commands or rows in a matrix; also separates columns in a matrix definition
[]	Forms a matrix
ans	Default MATLAB variable that stores calculation results
eps	Smallest distinguishable difference between two numbers
Inf	Infinity
NaN	Not a Number
pi	Default MATLAB variable that stores the value of π

EXERCISES

- Determine the value of y when $y(x)=e^x+x$ and $x = 4$.
- Determine the value of y when $y(x)=\cos(x+2)$ and $x = 3.4$.
- Determine the value of y when $y(x)=10e^{-5x}\sin(x)$ and $x = 3.2$.
- Determine the values of `realmin/realmax` and `realmax/realmin`.
- Determine how MATLAB handles
 - division of a scalar by zero.
 - division as an array operation (`./`) when the divisor is a vector or matrix that contains zeros.
 - division as a matrix operation (`/`) when the divisor contains some, but not all, zeros.
 - division as a matrix operation when the divisor contains all zeros.
- Create a matrix **M**, in which the first column contains the complex numbers $0+1i$, $3+4i$, $-3-4j$, $5+5i$, and $5-5i$. The second column contains the magnitudes (absolute values) of the numbers, and the third column contains the phase angles of the numbers.
- Use the quadratic formula to determine the roots of the quadratic polynomial $y(x) = x^2 - 1$.
- Use the quadratic formula to determine the roots of the quadratic polynomial $y(x) = x^2 - 2x + 2.5$.
- Create an X-Y plot of the linear function $y(x) = -4x + 4$ from $x = -4$ to $x = +4$.
- Create an X-Y plot of the linear function $y(x)=5x-3$ for x equals -10 to $+10$ in increments of 0.1 .
- Create a vector **V** containing 11 linearly spaced numbers, ranging from -5 to $+5$.

12. Create a vector **V** containing 6 logarithmically spaced numbers, ranging from 0.001 to 100.
13. Create two vectors, **Va** and **Vb**. **Va** contains the integers -4 to +4. **Vb** contains the squares of the corresponding values in **Va**. Compute **Va + Vb** and **|Va - Vb|**.
14. Create two vectors, **Va** and **Vb**. **Va** contains the integers -4 to +4. **Vb** contains the squares of the corresponding values in **Va**. Using array (element-by-element) operations, compute **VaVb** and **Va/Vb**.
15. Create the vector **x**, which contains the integers -4 to +4. Using an array (element-by-element) operation, compute 2^x .
16. Create the X-Y plot $y(x) = 2^x$, from $x = -4$ to $+4$. (Use an array operation to create the **y** vector.)
17. Plot the contents of a vector that is defined by $X_i = \sin(i/4)$, $i=0..100$ against the contents of a vector that is defined by $Y_i = \cos(i/8)$, $i=0..100$.
18. Create a 2-column matrix **TEMP**, in which the first column contains the integer Fahrenheit temperatures from 32 to 102, in 2-degree increments. The second column contains the Celsius equivalents of the first column values.
19. Create a 2-row matrix **ANGLE**, in which the first row contains the angles 0 to 90, in 10-degree increments. The second row contains the radian equivalents of the first row values.
20. Create a 3-column matrix **SPEED**, in which the first column contains speeds from 0 mph to 100 mph in 10-mph increments. The second column contains the km/hr equivalents of the first column. The third column contains the m/s equivalents of the first column.
21. Create a 3-row matrix in which the first row contains the range 0 to 2 in increments of $\pi/4$. Each value in the second row is the cosine of the corresponding value in the first row. Each value in the third row is the absolute value of the corresponding value in the second row.
22. Create two 3 x 3 matrices, **Ma** and **Mb**. **Ma** contains all 1s; **Mb** contains all 2s. Compute **Ma + Mb** and **Mb - Ma**.
23. Create two 3 x 3 matrices, **Ma** and **Mb**. **Ma** contains all 1s; **Mb** contains all 2s. Using array (element-by-element) operations, compute **Ma Mb** and **Mb/Ma**.
24. Create two 3 x 3 matrices, **Ma** and **Mb**. **Ma** contains all 1s; **Mb** contains, in the first row, 1, 2, 3; in the second row, 4, 5, 6; and in the third row, 7, 8, 0. Using *matrix* operations, compute **Ma Mb**, **Mb/Ma**, and **Ma/Mb**. Note that, if the determinant of the divisor equals zero, the matrix division operation results in infinity (see Chapter 4).
25. Create a 3-row matrix that includes the names John, Mary, and Ellen. Create a second 3-row matrix that contains the addresses 123 F St, 14 Elm Ave., and 1234 C Blvd. (Remember, you must buffer short rows with blanks so that all rows contain the same number of characters.) Combine the two matrices into one 3-row matrix and display the result, listing the names in one column and the addresses in a second column. Use the whos command to determine the size of the resulting matrix.

26. Plot the following data on one graph, using a different color for each curve. Label the graph and its axes appropriately. Assume the first row contains sampling times in seconds. The remaining three rows contain pressure data, in pounds per square inch, measured at three different points in a container.

Time:	1	2	3	4	5	6	7	8
Point a:	20	20	21	22	23	22	21	21
Point b:	18	19	17	18	19	18	18	17
Point c:	16	15	16	17	18	17	16	15

ANSWERS TO SUB-SECTION SELF-TESTS

Sub-Section Self-Test 2-1

- 1.1198
- 13.5556
- 0.2536

Sub-Section Self-Test 2-2

Hs = -2.0466

Sub-Section Self-Test 2-3

NaN
NaN

Sub-Section Self-Test 2-4

R1 = 0.0625 + 0.4961i
R2 = 0.0625 - 0.4961i

Sub-Section Self-Test 2-5

x=[-3, 3];
y=-2.1*x-7.2;
plot(x,y)
grid

Sub-Section Self-Test 2-6

V1=[1:100];
V2=[101:200];
V3=[V1(96:100) V2(1:5)]

Sub-Section Self-Test 2-7

Entering: lookfor average
Returns: MEAN Average or mean value.

Sub-Section Self-Test 2-8

X=[-1,1,2,4];
Y=X.^2+3.*X-1

Sub-Section Self-Test 2-9

Deflection = [1.1, 1.3, 0.1, -0.1, -1.1, -0.5,
0.0, 0.2, 0.4, 0.1, -0.2, -0.8, -0.1, 0.2, 0.3];
plot(Deflection)

Sub-Section Self-Test 2-10

X=[1 3.2; 2 4.4; 3 4.8; 4 3.8; 5 4.1; 6 3.9];
Y=[X(:,1)+X(:,2)]

Sub-Section Self-Test 2-11

M= [logspace(-2,0,7); linspace(1,100,7)]

Sub-Section Self-Test 2-12

X=[-2 -1 0 1; -1 0 1 2; 0 1
2 3];
Y=X.^2+2.*X-1

Sub-Section Self-Test 2-13

A=[1 2 3 4]; % creates a 1 x 4 row vector
B=[1;2;3;4]; % creates a 4 x 1 column
vector

A*B = 30

B*A =

1	2	3	4
2	4	6	8
3	6	9	12
4	8	12	16

A/B = A*(1/B), but the dimensions of A are 1 x 4, and the dimensions of (1/B) are 1 x 4; therefore they cannot be multiplied, and A/B cannot be computed. 1/A is undefined; therefore, B/A = B*(1/A) cannot be computed.

Sub-Section Self-Test 2-14

```
oil=['West Side '; 'Lower Kern '; 'Lost Hills
']; % all 3 entries have 11 characters
wells=['11-2'; '13-2'; '14-1']; % all 3 entries
have 4 characters
result=[oil, wells]
```

Sub-Section Self-Test 2-15

```
voltdata = [0 4.8900 4.8600 4.9900; 1.0000
4.7200 4.8200 4.8600;...
2.0000 4.6800 4.7000 4.7300; 3.0000 4.6200
4.6800 4.7100;...
4.0000 4.5800 4.6000 4.6300];
aves=[mean(voltdata(1,2:4)),
mean(voltdata(2,2:4)),
mean(voltdata(3,2:4)),...
mean(voltdata(4,2:4)),
mean(voltdata(5,2:4))];
plot(voltdata(:,1),aves)
```