BENHA UNIVERSITY

FACULTY OF SCIENCE

DEPARTMENT OF MATHEMATICS

An Introduction to MATLAB with Applications

PREPARED BY

Dr. Abdelhameed Mohamed Nagy (A. M. Nagy[1])

2013

[1]abdelhameed_nagy@yahoo.com

Contents

Chapter 1

Matlab Technical Computing Environment

The primary goal of this chapter is to help you to become familiar with the Matlab software, a powerful tool. It is particularly important to familiarize yourself with the user interface and some basic functionality of Matlab. To this end, it is worthwhile to at least work through the examples in this chapter (actually type them in and see what happens). Of course, it is even more useful to experiment with the principles discussed in this chapter instead of just sticking to the examples. The chapter is set up in such a way that it affords you time to do t

1.1 Purpose and Philosophy of Matlab

Matlab is a high-performance programming environment for numerical and technical applications. The first version was written at the University of New Mexico in the 1970s. The "MATrix LABoratory" program was invented by Cleve Moler to provide a simple and interactive way to write programs using the Linpack and Eispack libraries of FORTRAN subroutines for matrix manipulation. Matlab has since evolved to become an effective and powerful tool for programming, data visualization and analysis, education, engineering and research.

The strengths of Matlab include extensive data handling and graphics capabilities, powerful programming tools and highly advanced algorithms. Although it specializes in numerical computation, Matlab is also capable of performing symbolic computation by having an interface with Maple (a leading symbolic mathematics computing environment). Besides fast numerics for linear algebra and the availability of a large number of domain-specific built-in functions and libraries (e.g., for statistics, optimization, image processing, neural networks), another useful feature of Matlab is its capability to easily generate various kinds of visualizations of your data and/or simulation results.

For every Matlab feature in general, and for graphics in particular, the usefulness of Matlab is mainly based on the large number of built-in functions and libraries. The intention of this tutorial is not to provide a comprehensive coverage of all Matlab features but rather to prepare you for your own exploration of its functionality. The online help system is an immensely powerful tool in explaining the vast collection of functions and libraries available to you, and should be the most frequently used tool when programming in Matlab. Note that this tutorial will not cover any of the functions provided in any of the hundreds of toolboxes, since each toolbox is licensed separately and their availability to you can vary. We will indicate in each section if a particular toolbox is required. If you have additional toolboxes available to you, we recommend using the online help system to familiarize yourself with the additional functions provided. Another tool for help is the Internet. A quick online search will usually bring up numerous useful web pages designed by other Matlab users trying to help out each other.

As stated previously, Matlab is essentially a tool sophisticated one, but a tool nevertheless. Used properly, it enables you to express and solve computational and analytic problems from a wide variety of domains.

The Matlab environment combines computation, visualization, and programming around the central concept of the matrix. Almost everything in Matlab is represented in terms of matrices and matrix-manipulations.

Matlab provides a technical computing environment designed to support the implementation of computational tasks.

Briefly, Matlab is an interactive computing environment that enables numerical computation and data visualization.

Matlab has hundreds of built-in functions and can be used to solve problems ranging from the very simple to the sophisticated and complex. Whether you want to do some simple numerical or statistical calculations, some complex statistics, solve simultaneous equations, make a graph, or run and entire simulation program, Matlab can be an effective tool.

Matlab has proven to be extraordinarily versatile and capable in its ability to help solve problems in applied math, physics, chemistry, engineering, finance - almost any application area that deals with complex numerical calculations.

1.2 Workspace, Windows, and Help

Running Matlab

- Unix: From a terminal window, type Matlab, followed by the Enter key.
- Win95, Winxp and Win7: double-click on the Matlab icon or select Matlab from Start/Programs.

Display Windows

- Command window Enter commands and data, display results Prompt >> or EDU >>
- Graphics (Figure) window

  Display plots and graphs

  Created in response to graphics commands

- M-file editor/debugger window

  Create and edit scripts of commands called M-files

When you begin Matlab, the command window will be the active window. As commands are executed, appropriate windows will automatically appear; you can activate a window by clicking the mouse in it.

Getting Help

- help On-line help, display text at command line

  help, by itself, lists all help topics

  help topic provides help for the specified topic

  help command provides help for the specified command

  help help provides information on use of the help command

- helpwin - On-line help, separate window for navigation.
- helpdesk - Comprehensive hypertext documentation and troubleshooting
- demo - Run demonstrations
- intro - Interactive introduction to Matlab

Interrupting and Terminating Matlab

- Ctrl-C (pressing the Ctrl and c keys simultaneously): Interrupts (aborts) processing, but does not terminate Matlab. You may want to interrupt Matlab if you mistakenly command it to display thousands of results and you wish to stop the time-consuming display.

- quit: Terminates Matlab
- exit: Terminates Matlab
- Select Exit under File menu: Terminates Matlab (MS Windows)

1.3 Scalar Mathematics

Scalar mathematics involves operations on **single-valued** variables. Matlab provides support for scalar mathematics similar to that provided by a calculator. For more information, type help ops.
The most basic Matlab command is the mathematical **expression**, which has the following properties:

- Mathematical construct that has a value or set of values.
- Constructed from numbers, operators, and variables.
- Value of an expression found by typing the expression and pressing Enter

1.3.1 Numbers

Matlab represents numbers in two form, fixed point and floating point.
Fixed point: Decimal form, with an optional decimal point. For example:

1.2458 -255 0.00012

Floating point: Scientific notation, representing $m \times 10^e$
For example: 1.2458×10^4 is represented as 1.2458e4
It is called floating point because the decimal point is allowed to move. The number has two parts:

- mantissa m: fixed point number (signed or unsigned), with an optional decimal point (2.6349 in the example above)

- exponent e: an integer exponent (signed or unsigned) (4 in the example).
- Mantissa and exponent must be separated by the letter e (or E).

Scientific notation is used when the numbers are very small or very large. For example, it is easier to represent 0.0000000002 as 2e-9.

Matlab recognizes several different kinds of numbers

Table 1.1:

Type	Examples
Integer	1612, -1597
Real	1.242, -8.76
Complex	3.41 - 2.3i ($i = \sqrt{-1}$)
Inf	Infinity (result of dividing by 0)
NaN	Not a Number, 0/0

All computations in Matlab are done in double precision, which means about 15 significant digits. The format-how Matlab prints numbers-is controlled by the "format" command. Type help format for full list. Should you wish to switch back to the default format then format will suffice. Table 1.2 below indicates the affect of changing the display format for the variable rt.

Table 1.2: Display formats.

Command	Description	Example
format short	Fixed-point with 4 decimal digits	>> rt = 351/7 rt = 50.1429
format long	Fixed-point with 15 decimal digits	>> rt = 351/7 rt = 50.142857142857146
format short e	Fixed-point with 4 decimal digits	>> rt = 351/7 rt = 5.0143e+01
format long e	Fixed-point with 15 decimal digits	>> rt = 351/7 rt = 5.014285714285715e+01
format short g	Best of 5 digit fixed or floating point	>> rt = 351/7 rt = 50.143
format long g	Best of 15 digit fixed or floating point	>> rt = 351/7 rt = 50.1428571428571
format bank	Two decimal digits	>> rt = 351/7 rt = 50.14
format compact	Eliminates empty lines to allow more lines with information displayed on the screen	
format loose	Adds empty lines (opposite of compact)	

1.3.2 Operators

The evaluation of expressions is achieved with arithmetic *operators*, shown in the table below. Operators operate on *operands* (a and b in the table). Examples of expressions constructed from numbers and operators, processed by Matlab:

```
>> 2 + 5
ans =
    7

>> 5/10
ans =
    0.5000
```


Table 1.3:

Operation	Algebraic form	Matlab	Example
Addition	$a+b$	$a+b$	5 + 2
Subtraction	$a-b$	a - b	7 - 2
Multiplication	$a \times b$	a*b	3 * 4
Right division	$a \div b$	a/b	12/6
Left division	$b \div a$	$a\backslash b$	6\12
Exponentiation	a^b	a ^ b	3^2

Prompt >> is supplied by Matlab, indicates beginning of the command. ans = following the completion of the command with the Enter key marks the beginning of the answer.

Precedence of operations (order of evaluation)

Since several operations can be combined in one expression, there are rules about the order in which these operations are performed:

1. Parentheses, innermost first.
2. Exponentiation (^), left to right.
3. Multiplication (*) and division (/ or \) with equal precedence, left to right.
4. Addition (+) and subtraction (-) with equal precedence, left to right.

When operators in an expression have the same precedence the operations are carried out from left to right. Thus 4 /5*6 is evaluated as (4/5)*6 and not as 4 / (5 * 6).

```
>> (2/3^2*5)*(3-4^3)^2
ans =
   4.1344e+03
```

1.3.3 Variables and Assignment Statements

Variable names can be assigned to represent numerical values in Matlab.
Assignment statement: Matlab command of the form:

- *variable = number*
- *variable = expression*

When a command of this form is executed, the expression is evaluated, producing a number that is assigned to the variable. The variable name and its value are displayed. If a variable name is not specified, Matlab will assign the result to the default variable, ans, as shown in previous examples.

```
>> 3-2^4
ans =
    -13

>> ans*5
ans =
    -65
```

The result of the first calculation is labelled ans by Matlab and is used in the second calculation where its value is changed.
We can use our own names to store numbers:

```
>> x = 3-2^4
x =
  -13

>> y = x*5
y =
  -65
```


so that $x = 13$ and $y = -65$.

Special variables:

ans: default variable name

pi: ratio of circle circumference to its diameter, $pi = 3.1415926...$

eps: smallest amount by which two numbers can differ

inf or Inf : infinity, e.g. $1/0$

nan or NaN : not-a-number, e.g. $0/0$

date: current date in a character string format, such as 1-September-2012.

flops: count of floating-point operations.

Commands involving variables:

who: lists the names of defined variables

whos: lists the names and sizes of defined variables

clear: clears all variables, resets default values of special variables

clear *var*: clears variable *var*

clc: clears the command window, homes the cursor (moves the prompt to the top line), but does not affect variables.

clf: clears the current figure and thus clears the graph window.

more on: enables paging of the output in the command window.

more off: disables paging of the output in the command window.

When more is enabled and output is being paged, advance to the next line of output by pressing Enter; get the next page of output by pressing the spacebar. Press q to exit out of displaying the current item.

Punctuation and Comments

- Semicolon (;) at the end of a command suppresses the display of the result

- Commas and semicolons can be used to place multiple commands on one line, with commas producing display of results, semicolons supressing
- Percent sign (%) begins a comment, with all text up to the next line ignored by Matlab
- Three periods (...) at the end of a command indicates that the command continues on the next line. A continuation cannot be given in the middle of a variable name.

```
>> x=-13; y = 5*x, z = x^2+y
y =
  -65
z =
  104
>> who
Your variables are:
x  y  z
>> whos
  Name      Size            Bytes  Class     Attributes

  x         1x1                 8  double
  y         1x1                 8  double
  z         1x1                 8  double
```

1.4 Basic Mathematical Functions

Matlab supports many mathematical functions, most of which are used in the same way you write them mathematically.

Elementary math functions (enter help elfun for a more complete list):

abs(x) Absolute value $|x|$

sign(x) Sign, returns -1 if $x < 0$, 0 if $x = 0$, 1 if $x > 0$

exp(x) Exponential e^x

log(x) Natural logarithm lnx

log10(x) Common (base 10) logarithm $log_{10}x$

sqrt(x) Square root $\sqrt{x}$

round(x) Round to the nearest integer. For example, round(17/5) is 3.

fix(x) Round toward zero. For example, fix(13/5) is 2.

ceil(x) Round toward infinity. For example, ceil(11/5) is 3.

floor(x) Round toward minus infinity. For example, floor(-9/4) is -3.

rem(x,y) Remainder of x/y. For example, rem(72,5) is 2. Also called the **modulus** function.

Information about these functions is displayed by the command help function. For example:

```
>> help sqrt

SQRT Square root.
   SQRT(X) is the square root of the elements of X. Complex
   results are produced if X is not positive.

   See also SQRTM.
```

Matlab contains also a number of functions for performing computations which require the use of logarithms, trigonometric and hyperbolic math functions see Table 1.4.

Table 1.4: Trigonometric and hyperbolic functions.

Function	Description
sin(x)	Computes the sine of x, where x is in radians.
cos(x)	Computes the cosine of x, where x is in radians
tan(x)	Computes the tangent of x, where x is in radians.
asin(x)	Computes the arcsine or inverse sine of x, where x must be between 1 and 1.
acos(x)	Computes the arccosine or inverse cosine of x, where x must be between 1 and 1.
atan(x)	Computes the arctangent or inverse tangent of x.
atan2(y,x)	Computes the arctangent or inverse tangent of the value y/x.
sinh(x)	Computes the hyperbolic sine of x, which is equal to $\frac{e^x - e^{-x}}{2}$.
cosh(x)	Computes the hyperbolic cosine of x, which is equal to $\frac{e^x + e^{-x}}{2}$.
tanh(x)	Computes the hyperbolic tangent of x, which is equal to $\frac{\sinh(x)}{\cosh(x)}$.
asinh(x)	Computes the inverse hyperbolic sine of x, which is equal to $\ln\left(x + \sqrt{x^2+1}\right)$.
acosh(x)	Computes the inverse hyperbolic cosine of x, which is equal to $\ln\left(x + \sqrt{x^2-1}\right)$.
atanh(x)	Computes the inverse hyperbolic tangent of x, which is equal to $\ln\sqrt{\frac{x+1}{x-1}}$, for $\lvert x\rvert \leq 1$.

Displaying Values and Text

There are three ways to display values and text in `Matlab`, to be described in this section:

1. By entering the variable name at the `Matlab` prompt, without a semicolon.
2. By use of the command disp.
3. By use of the command fprintf.

- From the prompt: As demonstrated in previous examples, by entering a variable name, an assignment statement, or an expression at the `Matlab` prompt, without a semicolon, the result will be displayed, proceeded by the variable name (or by ans if only an expression was entered). For example:

```
>> temp = 39
temp =
     39
```

- disp: There are two general forms of the command disp that are useful in displaying results and annotating them with units or other information:

  1. disp(*variable*): Displays value of variable without displaying the variable name.

  2. disp(*string*): Displays string by stripping off the single quotes and echoing the characters between the quotes.

String: A group of keyboard characters enclosed in single quote marks (’). The quote marks indicate that the enclosed characters are to represent ASCII text.

```
>> temp = 39;
>> disp(temp); disp('degrees C')
   39
degrees C
```

Note that the two disp commands were entered on the same line so that they would be executed together.

- fprintf One of the weaknesses of Matlab is its lack of good facilities for formatting output for display or printing. A function providing some of the needed capability is fprintf. This function is similar to the function of the same name in the ANSI C language, so if you are familiar with C, you will be familiar with this command. The fprintf function provides more control over the display than is provided with disp. In addition to providing the display of variable values and text, it can be used to control the format to be used in the display, including the specification to skip to a new line. The general form of this command is:

```
fprintf('format string', list of variables)
```

The format string contains the text to be displayed (in the form of a character string enclosed in single quotes) and it may also contain format specifiers to control how the variables listed are embedded in the format string. The format specifiers include:

%*w*.*d*`f` Display as fixed point or decimal notation (defaults to short), with a width of *w* characters (including the decimal point and possible minus sign, with *d* decimal places. Spaces are filled in from the left if necessary. Set *d* to 0 if you don't want any decimal places, for example %5.0`f`. Include leading zeros if you want leading zeroes in the display, for example %06.0`f`.

%*w*.*d*`e` Display using scientific notation (defaults to short *e*), with a width of *w* characters (including the decimal point, a possible minus sign, and five for the exponent), with *d* digits in the mantissa after the decimal point. The mantissa is always adjusted to be less than 1.

%*w*.*d*`g` Display using the shorter of tt short or short *e* format, with width w and *d* decimal places.

`\n` Newline (skip to beginning of next line)

The *w*.*d* width specifiers are optional. If they are left out, default values are used. Examples:

```
>> fprintf('The temperature is %f degrees F \n', temp)
The temperature is 39.000000 degrees C

>> fprintf(The temperature is %4.1f degrees F \n, temp)
The temperature is 39.0 degrees C
```

Chapter 2

Arrays

2.1 Arrays

An array is a list of numbers arranged in rows and/or columns. A one-dimensional array is a row or a column of numbers and a two-dimensional array has a set of numbers arranged in rows and columns. An array operation is performed element-by-element.

2.1.1 Row Vector

A vector is a row or column of elements.
In a row vector, the elements are entered with a space or a comma between the elements inside the square brackets. For example, $x = [7 - 12 - 58]$.

```
>> x = [7 -1 2 -5 8]

x =

     7    -1     2    -5     8
```

2.1.2 Column Vector

In a column vector, the elements are entered with a semicolon between the elements inside the square brackets. For example, $x = [7; -1; 2; -5; 8]$.


```
>>  x = [7; -1; 2; -5; 8]

x =

     7
    -1
     2
    -5
     8
```

2.1.3 Matrix

A matrix is a two-dimensional array which has numbers in rows and columns. A matrix is entered row-wise with consecutive elements of a row separated by a space or a comma, and the rows separated by semicolons or carriage returns. The entire matrix is enclosed within square brackets. The elements of the matrix may be real numbers or complex numbers. For example, to enter the matrix,

$$A = \begin{pmatrix} 2 & 4 & 3 \\ -1 & 0 & 1 \\ 2 & 3 & 4 \end{pmatrix}$$

The MATLAB input command is

```
>>   A = [2 4 3;-1 0 1;2 3 4]

A =

     2     4     3
    -1     0     1
     2     3     4
```

2.1.4 Addressing Arrays

A colon can be used in Matlab to address a range of elements in a vector or a matrix.

2.1.5 Colon notation:

Addresses a block of elements
The format for colon notation is:

(start:increment:end)

where start is the starting index, increment is the amount to add to each successive index, and end is the ending index, where start, increment, and end must be integers. The increment can be negative, but negative indices are not allowed to be generated. If the increment is to be 1, a shortened form of the notation may be used:

(start:end)

```
>> y = 0:2:10

y =

     0     2     4     6     8    10

>> x = 1:7

x =

     1     2     3     4     5     6     7
```


2.1.6 The linspace and logspace Functions

⋆ linspace: This function generates a vector of uniformly incremented values, but instead of specifying the increment, the number of values desired is specified. This function has the form:

linspace(start,end,number)

The increment is computed internally, having the value:

$$\text{increment} = \frac{\text{end} - \text{start}}{\text{number} - 1}$$

For example:

```
>> x=linspace(0,pi,11)
x =

  Columns 1 through 7

    0  0.3142  0.6283  0.9425  1.2566  1.5708  1.8850

  Columns 8 through 11

    2.1991 2.5133  2.8274  3.1416
```

In this example:

$$\text{increment} = \frac{\pi}{11 - 1} = 0.1\pi$$

⋆ logspace: This function generates logarithmically spaced values and has the form:

logspace(start_exponent,end_exponent,number)

To create a vector starting at $10^0 = 1$, ending at $10^2 = 100$ and having 11 values.

```
>> logspace(0,2,11)

ans =

  Columns 1 through 7

    1.0000   1.5849   2.5119   3.9811   6.3096   10.0000   15.8489

  Columns 8 through 11

   25.1189   39.8107   63.0957   100.0000
```

Colon for a vector

$\mathbf{V}$(:) refers to all the elements of the vector $\mathbf{V}$ (either a row or a column vector).
$\mathbf{V}$(m:n) refers to elements m through n of the vector $\mathbf{V}$.
For instance,

```
>> V = [2 5 -1 11 8 4 7 3 11];
>> u = V (2:8)    % 2:8 means ''start with 2 and count up to 8.''

u =

     5     -1     11      8      4      7      3
>> w = V (4:end)  % 4:end means ''start with 4 and count up
                  % to the end of the vector.''
```



```
w =

    11    8    4    7    3   11

>> V(3:-1:1)   % 3:-1:1 means ''start with 3 and count
               % down to 1.''

ans =

    -1    5    2

>> V(2:2:7)   % 2:2:7 means ''start with 2, count up by 2,
              % and stop at 7.''

ans =

     5    11    4
```

Colon for a matrix

Table 2.1 gives the use of a colon in addressing arrays in a matrix.

Table 2.1: Colon use for a matrix.

Command	Description
$\mathbf{A}(:, n)$	Refers to the elements in all the rows of a column n of the matrix A.
$\mathbf{A}(n, :)$	Refers to the elements in all the columns of row n of the matrix A.
$\mathbf{A}(:, m : n)$	Refers to the elements in all the rows between columns m and n of the matrix A.
$\mathbf{A}(m : n, :)$	Refers to the elements in all the columns between rows m and n of the matrix A.
$\mathbf{A}(m : n, p : q)$	Refers to the elements in rows m through n and columns p through q of the matrix A.

2.2 Adding Elements to a Vector or a Matrix

A variable that exists as a vector or a matrix can be changed by adding elements to it. Addition of elements is done by assigning values of the additional elements, or by appending existing variables. Rows and/or columns can be added to an existing matrix by assigning values to the new rows or columns.

```
>> S = [1 3 5 7]

S =

     1     3     5     7

>> S(6) = -1

S =

     1     3     5     7     0    -1

>> A=[1 2 3;2 -1 4;0 2 1]

A =

     1     2     3
     2    -1     4
     0     2     1

>> A(4,1) = 5
```



```
A =

     1     2     3
     2    -1     4
     0     2     1
     5     0     0

>> A(1,7) = 7

A =

     1     2     3     0     0     0     7
     2    -1     4     0     0     0     0
     0     2     1     0     0     0     0
     5     0     0     0     0     0     0
```

2.3 Deleting Elements

An element or a range of elements of an existing variable can be deleted by reassigning blanks to these elements. This is done simply by the use of square brackets with nothing typed in between them.

```
>> t = [1 3 5 -1 0 5]

t =

     1     3     5    -1     0     5

>> t(2) = []
```

```
t =

     1     5    -1     0     5

>> B=[1 2 3;2 -1 4;0 2 1]

B =

     1     2     3
     2    -1     4
     0     2     1

>> B(1,:) = []

B =

     2    -1     4
     0     2     1
```

2.4 Built-in Functions

Some of the built-in functions available in `Matlab` for managing and handling arrays as listed in Table 5.4.

Table 2.2: Built-in functions for handling arrays.

Function	**Description**	**Example**
length(A)	Returns the number of elements in the vector A.	>> A = [5 9 2 4]; >> length(A) ans = 4
		Continued on next page

Table 2.2 – continued from previous page

Function	Description	Example
size(A)	Returns a row vector $[m, n]$ where m and n are the size $m \times n$ of the array A.	>> A = [2 3 0 8 11;6 17 5 7 1] A = 2 3 0 8 11 6 17 5 7 1 >> size(A) ans = 2 5
reshape(A, m, n)	Rearrange a matrix A that has r rows and s columns to have m rows and n columns. r times s must be equal to m times n.	>> A = [3 1 4;9 0 7] A = 3 1 4 9 0 7 >> B = reshape(A, 3, 2) B = 3 0 9 4 1 7
diag(v)	When v is a vector, creates a square matrix with the elements of v in the diagonal	>> v = [3 2 1]; >> A = diag(v) A = 3 0 0 0 2 0 0 0 1
diag(A)	When A is a matrix, creates a vector from the diagonal elements of A.	>> A = [1 8 3;4 2 6;7 8 3] >> A = 1 8 3 4 2 6 7 8 3 >> ve = diag(A) ve = 1 2 3

2.5 Operations With Arrays

We consider here matrices that have more than one row and more than one column.

2.5.1 Identity Matrix

An identity matrix is a square matrix in which all the diagonal elements are 1's, and the remaining elements are 0's. If a matrix A is square, then it can be multiplied by the identity matrix, I, from the left or from the right:

$$AI = IA = A$$

To generate an identity matrix in Matlab:

eye(n) Returns an $n \times n$ identity matrix.

eye(m, n) Returns an $m \times n$ matrix with ones on the main diagonal and zeros elsewhere.

eye(size(a)) Returns a matrix with ones on the main diagonal and zeros elsewhere that is the same size as A.

```
>> I = eye(3)

I =

     1     0     0
     0     1     0
     0     0     1

>> I = eye(3,2)

I =
```



```
     1     0
     0     1
     0     0
```

2.5.2 Addition and Subtraction of Matrices

The addition (the sum) or the subtraction (the difference) of the two arrays is obtained by adding or subtracting their corresponding elements. These operations are performed with arrays of identical size (same number of rows and columns).

For example, if A and B are two arrays (2×3 matrices).

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix}$$

Then,

$$A + B = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & a_{13} + a_{13} \\ a_{21} + b_{21} & a_{22} + b_{22} & a_{23} + b_{31} \end{pmatrix}.$$

```
>> A= [1 2 3;0 1 2;2 5 7]

A =

     1     2     3
     0     1     2
     2     5     7

>> B = [0 1 2;-1 2 3;1 2 3]

B =

     0     1     2
    -1     2     3
```

```
     1     2     3
```

```
>> A + B
```

```
ans =

     1     3     5
    -1     3     5
     3     7    10
```

2.5.3 Array Multiplication

The value in position $c_{i,j}$ of the product C of two matrices, A and B, is

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

```
>> A= [1 2 3;0 1 2;2 5 7];
>> B = [0 1 2;-1 2 3;1 2 3];
>> C = A*B
```

```
C =

     1    11    17
     1     6     9
     2    26    40
```

2.5.4 Transpose

The transpose of a matrix is a new matrix in which the rows of the original matrix are the columns of the new matrix. The transpose of a given matrix A is denoted by A^T. In `Matlab`, the transpose of the matrix A is denoted by A'.


```
>> A= [1 2 3;0 1 2;2 5 7];
>> A_1 = A'
```

```
A_1 =

     1     0     2
     2     1     5
     3     2     7
```

2.5.5 Determinant

A determinant is a scalar computed from the entries in a square matrix. For a 2×2 matrix A, the determinant is

$$|A| = a_{11}a_{22} - a_{21}a_{12}$$

`Matlab` will compute the determinant of a matrix using the `det` function:
`det`(A): Computes the determinant of a square matrix A.

```
>> A= [1 2 3;0 1 2;2 5 7];
>> det(A)
```

```
ans =

    -1
```

2.5.6 Inverse of a Matrix

The matrix B is the inverse of the matrix A when the two matrices are multiplied and the product is an identity matrix. Both matrices A and B must be square and the order of multiplication can be AB or BA.

$$AB = BA = I$$

To generate the inverse of the matrix in Matlab, we use inv(A)

```
>> A = [1 0 3;-1 2 5;2 4 5]

A =

     1     0     3
    -1     2     5
     2     4     5

>> inv(A)

ans =

    0.2941   -0.3529    0.1765
   -0.4412    0.0294    0.2353
    0.2353    0.1176   -0.0588
```

2.6 Sets of Linear Equations

Consider the square system of n linear equations with n unknowns $x_1, \ldots, x_n$,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= y_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= y_2, \\ &\cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= y_n, \end{aligned}$$

or more compactly

$$Ax = y, \quad \text{where } A \text{ is } n \times n, \quad x \text{ and } y \text{ are } n \times 1. \tag{2.1}$$

Here

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n1} & \cdots & a_{nn} \end{pmatrix}, \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}.$$

We already met the matrix function inv. This function computes the inverse of a non-singular matrix. One solution would be therefore to write simply x = inv(A)*b. Since inverting a matrix is in fact equivalent to solving n times a system of n linear equations with the n columns of the identity matrix playing the role of the right hand sides y, it is clear that inversion of A is usually too expensive in terms of computer time. Matlab can solve the system in one statement: x=A \ y as we will see later.

2.6.1 Array Division

Matlab has two types of array division, which are the left division and the right division.

Left Division

The left division is used to solve the matrix equation $Ax = b$ where x and b are column vectors. Multiplying both sides of this equation by the inverse of A, A^1, we have

$$\begin{aligned} A^{-1}Ax &= A^{-1}b; \\ Ix &= A^{-1}b; \\ x &= A^{-1}b \end{aligned}$$

In Matlab, the above equation is written by using the left division character:

$$x = A\backslash b$$

```
>> A= [1 2 3;0 1 2;2 5 7];
>> b = [1 2 3]'

b =

     1
     2
     3

>> A\b

ans =

    -2
     0
     1

>> inv(A)*b

ans =

    -2
     0
     1
```

Right Division

The right division is used to solve the matrix equation $xA = b$ where x and b are row vectors. Multiplying both sides of this equation by the

inverse of A, A^1, we have

$$xAA^{-1} = bA^{-1};$$
$$x = bA^{-1}$$

In `Matlab`, this equation is written by using the right division character:

$$x = b/A$$

```
>> A= [1 2 3;0 1 2;2 5 7];
>> b = [1 2 3]

b =

     1     2     3

>> b/A

ans =

     1     0     0
```

2.7 Eigenvalues and Eigenvectors

Consider the following equation:

$$Ax = \lambda x, \tag{2.2}$$

where A is an $n \times n$ square matrix, x is a column vector with n rows and λ is a scalar.
The values of λ for which x are non-zero are called the eigenvalues of the matrix A, and the corresponding values of x are called the eigenvectors of

the matrix A.

Equation (2.2) can also be used to find the following equation:

$$(A - \lambda I)x = 0, \tag{2.3}$$

where I is an $n \times n$ identity matrix. Equation (2.3) corresponding to a set of homogeneous equations and has non-trivial solutions only if the determinant is equal to zero, or

$$|A - \lambda I|x = 0, \tag{2.4}$$

Equation (2.4) is known as the characteristic equation of the matrix A. The solution to Eq.(2.4) gives the eigenvalues of the matrix A.

`Matlab` determines both the eigenvalues and eigenvectors for a matrix A using the command eig(A).

eig(A): Computes a column vector containing the eigenvalues of A.

[V, D] = eig(A): Computes a square matrix V containing the eigenvectors of A as columns and a square matrix D containing the eigenvlaues (λ) of A on the diagonal.

```
>> A=[1 2 -1;2 0 3;1 2 3]

A =

     1     2    -1
     2     0     3
     1     2     3

>> eig(A)

ans =
```



```
    4.5456
    1.6231
   -2.1687

>> [V,D] = eig(A)

V =

   -0.0973   -0.8256    0.5673
   -0.5777    0.0247   -0.7990
   -0.8104    0.5637    0.1994


D =

    4.5456         0         0
         0    1.6231         0
         0         0   -2.1687
```

Triangular Factorization or Lower-Upper Factorization

Triangular or lower-upper factorization expresses a square matrix as the product of two triangular matricesa lower triangular matrix and an upper triangular matrix. The lu function in `Matlab` computes the **LU** factorization.

[L, U] = lu(A): Computes a permuted lower triangular factor in L and an upper triangular factor in U such that the product of L and U is equal to A.

The solution to $Ax = b$ is obtained with matrix division

$$x = A^{-1}b = U^{-1}L^{-1}b.$$

```
>> A=[1 2 -1;2 0 3;1 2 3]

A =

     1     2    -1
     2     0     3
     1     2     3

>> b=[1 2 3]'

b =

     1
     2
     3

>> [L,U] = lu(A)

L =

    0.5000    1.0000         0
    1.0000         0         0
    0.5000    1.0000    1.0000


U =

    2.0000         0    3.0000
         0    2.0000   -2.5000
```



```
         0         0    4.0000

>> x = inv(U)*inv(L)*b

x =

    0.2500
    0.6250
    0.5000
```

QR Factorization

The QR factorization method factors a matrix A into the product of an orthonormal matrix and an upper-triangular matrix. The `qr` function is used to perform the QR factorization in `Matlab`.

`[Q, R] = qr(A)`: Computes the values of Q and R such that $A = QR$.

Q will be an orthonormal matrix, and R will be an upper triangular matrix.

As LU factorization, we can use QR in solving the system $Ax = b$ as follows $x = R^{-1}Q^{-1}b$.

Note that: for a matrix A of size $m \times n$, the size of Q is $m \times m$, and the size of R is $m \times n$.

```
>> A=[1 2 -1;2 0 3;1 2 3]

A =

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

```
>> b=[1 2 3]'

b =

     1
     2
     3

>> [Q,R] = qr(A)

Q =

   -0.4082    0.5774   -0.7071
   -0.8165   -0.5774   -0.0000
   -0.4082    0.5774    0.7071


R =

   -2.4495   -1.6330   -3.2660
         0    2.3094   -0.5774
         0         0    2.8284

>> x = inv(R)*inv(Q)*b

x =

    0.2500
    0.6250
```



```
    0.5000
```

2.8 Element-By-Element Operations

Element-by-element operations can only be done with arrays of the same size. Element-by-element multiplication, division and exponentiation of two vectors or matrices is entered in `Matlab` by typing a period in front of the arithmetic operator. Table 2.3 lists these operations.

Table 2.3: Element-by-element operations

Arithmetic operators			
Matrix operators		Array operators	
+	Addition	+	Addition
-	Subtraction	-	Subtraction
*	Multiplication	.*	Array multiplication
^	Exponentiation	.^	Array exponentiation
/	Right division	./	Array right division
\	Left division	.\	Array left division

```
>> A=[1 2 -1;0 1 1;2 1 3];
>> B=[2 3 -1;1 2 4;1 2 4];
>> A.*B

ans =

     2     6     1
     0     2     4
     2     2    12

>> A.^2

ans =
```

```
     1     4     1
     0     1     1
     4     1     9

>> A./B

ans =

    0.5000    0.6667    1.0000
         0    0.5000    0.2500
    2.0000    0.5000    0.7500

>>  A.\B


ans =

    2.0000    1.5000    1.0000
       Inf    2.0000    4.0000
    0.5000    2.0000    1.3333

>> b = [1 0 3]'

b =

     1
     0
     3
```



```
>> A.*b
Error using  .*
Matrix dimensions must agree.

>> A*b

ans =

    -2
     3
    11
```

2.8.1 Built-in Functions for Arrays

Table 2.4 lists some of the many built-in functions available in Matlab for analysing arrays.

Table 2.4: Matlab built-in array functions

Function	Description	Example
mean(A)	If A is a vector, returns the mean value of the elements	>> A = [3 7 2 16]; >> mean(A) = ans = 7
[d, n] = max(A)	If A is a vector, d is the largest element in A, n is the position of the element (the first if several have the max value).	>> A = [3 7 2 16 9 5 18 13 0 4]; >> [d, n] = max(A) d = 18 n = 7
[d, n] = min(A)	The same as $[d, n] = max(A)$, but for the smallest element.	>> A = [3 7 2 16 9 5 18 13 0 4]; >> [d, n] = min(A) d = 0 n = 9
sum(A)	If A is a vector, returns the sum, of the elements of the vector.	>> A = [3 7 2 16]; >> sum(A) ans = 28
sort(A)	If A is a vector, arranges the elements of the vector in. ascending order.	>> A = [3 7 2 16]; >> sort(A) ans = 2 3 7 16
median(A)	If A is a vector, returns the median value of the elements of. the vector	>> A = [3 7 2 16]; >> median(A) ans = 5
std(A)	If A is a vector, returns the standard deviation of the. elements of the vector.	>> A = [3 7 2 16]; >> std(A) ans = 6.3770

2.9 Random Numbers Generation

There are many physical processes and engineering applications that require the use of random numbers in the development of a solution.
Matlab has two commands rand and randn that can be used to assign random numbers to variables.
The rand command: The rand command generates uniformly distributed over the interval [0, 1]. The command can be used to assign these numbers to a scalar, a vector or a matrix as shown in Table 2.5.

Table 2.5: The rand command

Function	Description	Example
rand	Generates a single random number between 0 and 1.	>> rand >> ans = 0.8147
rand(1, n)	Generates an n elements row vector of random numbers between 0 and 1.	>> a = rand(1,3) >> a = 0.9058 0.1270 0.9134
rand(n)	Generates an $n \times n$ matrix with random numbers between 0 and 1.	>> b = rand(3) >> b = 0.6324 0.5469 0.1576 0.0975 0.9575 0.9706 0.2785 0.9649 0.9572
rand(m,n)	Generates an $m \times n$ matrix with random numbers between 0 and 1.	>> c = rand(2,3) >> c = 0.4854 0.1419 0.9157 0.8003 0.4218 0.7922

2.9.1 The Random Command

Matlab will generate Gaussian values with a mean of zero and a variance of 1.0 if a normal distribution is specified. The Matlab functions for generating Gaussian values are as follows:
randn(n): Generates an $n \times n$ matrix containing Gaussian (or normal)

random numbers with a mean of 0 and a variance of 1.

`randn(m, n)`: Generates an $m \times n$ matrix containing Gaussian (or normal) random numbers with a mean of 0 and a variance of 1.

Chapter 3

Introduction to Graphics

A picture, it is said, is worth a thousand words. Matlab has a powerful graphics system for presenting and visualizing data, which is reasonably easy to use. (Most of the figures in this book have been generated by Matlab.)

This chapter introduces Matlab's high-level 2-D and 3-D plotting facilities. Low level features, such as handle graphics, are discussed later in this chapter.

3.1 Basic 2-D Graphs

Graphs (in 2-D) are drawn with the plot statement. In its simplest form, it takes a single vector argument as in plot(y). Matlab allows graphs to be created quickly and conveniently. For example, to create a graph of the t and $v = exp(t)$ arrays, we enter

```
>> t = linspace(1,10,20);
>> v = exp(t);
>> plot(t,v)
```

Axes are automatically scaled and drawn to include the minimum and maximum data points.

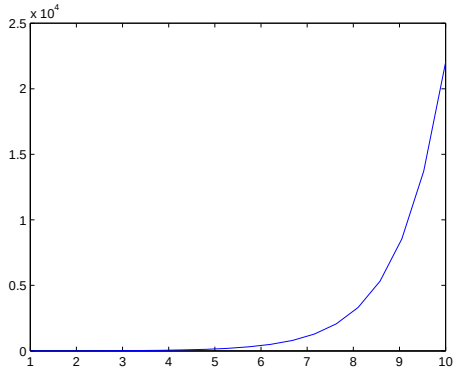
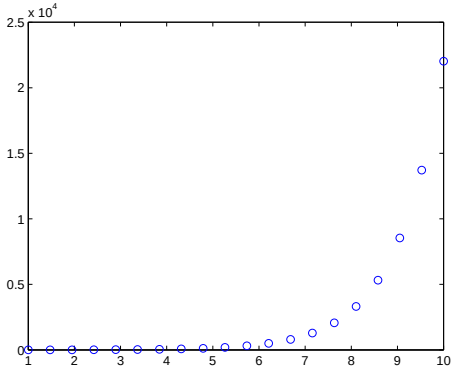


Table 3.1: Specifiers for colors, symbols, and line types.

Colors		Symbols		Line Types	
Blue	b	Point	·	Solid	-
Green	g	Circle	o	Dotted	:
Red	r	X-mark	x	Dashdot	-.
Cyan	c	Plus	+	Dashed	–
Magenta	m	Star	*		
Yellow	y	Square	s		
Black	k	Diamond	d		
White	w	Triangle(down)			
		Triangle(up)			
		Triangle(left)	<		
		Triangle(right)	>		
		Pentagram	p		
		Hexagram	h		

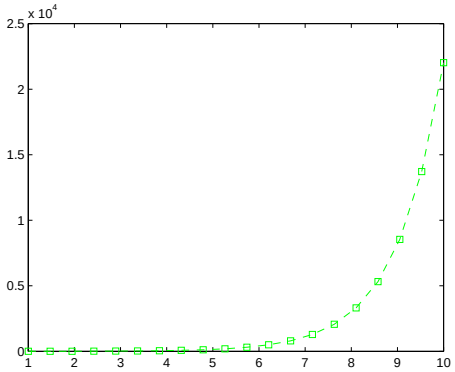
The plot command displays a solid thin blue line by default. If you want to plot each point with a symbol, you can include a specifier enclosed in single quotes in the plot function. Table 3.1 lists the available specifiers. For example, if you want to use open circles enter

```
>> plot(t, v, 'o')
```



You can also combine several specifiers. For example, if you want to use square green markers connected by green dashed lines, you could enter

```
>> plot(t, v, 's--g')
```

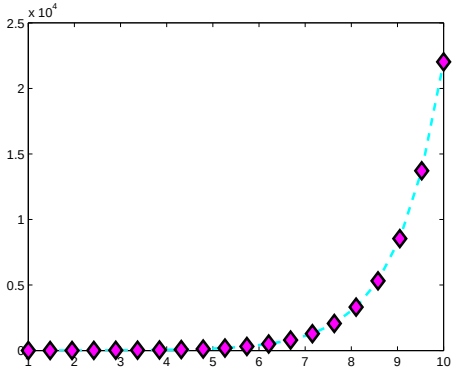


You can also control the line width as well as the markers size and its edge and face (i.e., interior) colors. For example, the following command uses a heavier (2-point), dashed, cyan line to connect larger (10-point) diamond-shaped markers with black edges and magenta faces:

```
>> plot(t,v,'--dc','LineWidth',2,...
```



```
'MarkerSize',10,...
'MarkerEdgeColor','k',...
'MarkerFaceColor','m')
```



Note that the default line width is 1 point. For the markers, the default size is 6 point with blue edge color and no face color.

Matlab allows you to display more than one data set on the same plot. For example, an alternative way to connect each data marker with a straight line would be to type

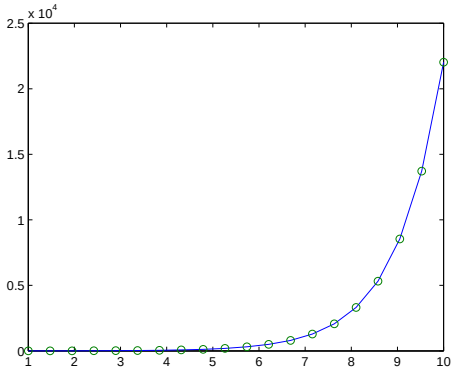
```
>> plot(t, v, t, v, 'o')
```

3.1.1 Labels

Graphs may be labeled with the following statements:

```
>> gtext('text')
```

writes a string ('text') in the graph window. gtext puts a cross-hair in the graph window and waits for a mouse button or keyboard key to be pressed. The cross-hair can be positioned with the mouse or the arrow keys.



Text may also be placed on a graph interactively with Tools $->$ Edit Plot from the figure window.

```
>> grid
```

adds/removes grid lines to/from the current graph. The grid state may be toggled.

```
>> text(t, v, 'text')
```

If t and v are vectors, the text is written at each point. If the text is an indexed list, successive points are labeled with corresponding rows of the text.

```
>> title('text')
```

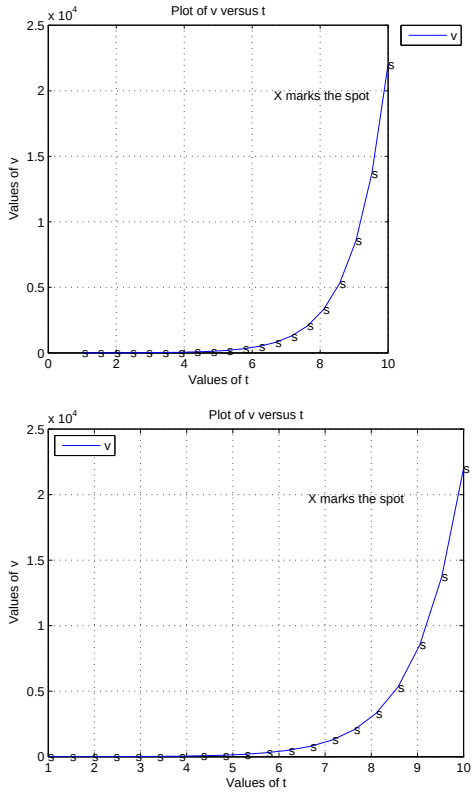
writes the text as a title on top of the graph.

```
>> xlabel('horizontal')
labels the x-axis.

>> ylabel('vertical')
labels the y-axis.

>>legend(string1,string2, ..... ,pos)
```


The legend command places a legend on the plot. The legend shows a sample of the line type of each graph that is plotted, and places a label, specified by the user, beside the line sample.
The strings are the labels that are placed next to the line sample. Their order corresponds to the order in which the graphs were created. The pos is an optional number that specifies where in the figure the legend is to be placed. The options are:

- pos = -1 Places the legend outside the axes boundaries on the right side.
- pos = 0 Places the legend inside the axes boundaries in a location that interferes the least with the graphs.
- pos = 1 Places the legend at the upper-right corner of the plot (default).
- pos = 2 Places the legend at the upper-left corner of the plot.
- pos = 3 Places the legend at the lower-left corner of the plot.
- pos = 4 Places the legend at the lower-right corner of the plot.

```
>> t = linspace(1,10,20);
>> v = exp(t);
>> plot(t, v)
>> text(t, v, 's')
>> title('Plot of v versus t')
>> xlabel('Values of t')
>> ylabel('Values of v')
>> gtext( 'X marks the spot')
>> grid
>> legend('v',-1)  % left figure
>> legend('v',2)   % right figure
```



3.1.2 Multiple plots on the same axes

There are at least two ways of drawing multiple plots on the same set of axes (which may however be rescaled if the new data falls outside the range of the previous data).

1- The easiest way is simply to use hold to keep the current plot on the axes. All subsequent plots are added to the axes until hold is released, either with hold off , or just hold, which toggles the hold state.

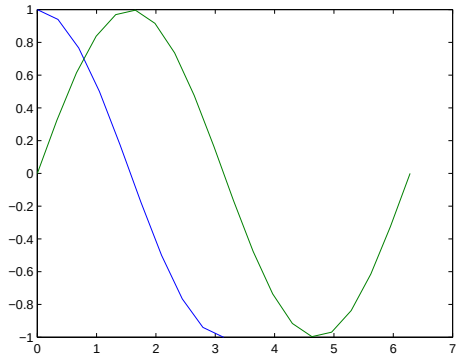
```
>> plot(t, v)
>> hold on
>> plot(t, v, 'o')
>> hold off
```


2- The second way is to use plot with multiple arguments, e.g.

```
>> plot(x1, y1, x2, y2, x3, y3, ... )
```

plots the (vector) pairs (x1, y1), (x2, y2), etc. The advantage of this method is that the vector pairs may have different lengths. Matlab automatically selects a different color for each pair.

```
>> t = linspace(0,pi,10);
>> v = cos(t);
>> t1 = linspace(0,2*pi,20);
>> v1 = sin(t1);
>> plot(t, v,t1,v1)
```



3.1.3 Axis limits

Whenever you draw a graph with Matlab it automatically scales the axis limits to fit the data. You can override this with

```
>> axis( [xmin, xmax, ymin, ymax] )
```

which sets the scaling on the *current* plot, i.e. draw the graph first, then reset the axis limits.

If you want to specify one of the minimum or maximum of a set of axis limits, but want MATLAB to autoscale the other, use Inf or -Inf for the autoscaled limit.

You can return to the default of automatic axis scaling with

```
>> axis auto
```

3.1.4 Multiple plots in a figure: subplot

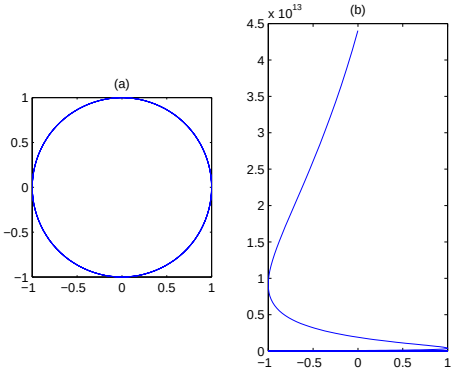
You can show a number of plots in the same figure window with the subplot function. It looks a little curious at first, but its quite easy to get the hang of it.
The statement

```
>> subplot(m, n, p)
```

divides the figure window into $m \times n$ small sets of axes, and selects the pth set for the current plot (numbered by row from the left of the top row). For example, the following statements produce the four plots shown in Figure 3.1

```
>> t = 0:pi/50:10*pi;
>> subplot(1,2,1);plot(sin(t),cos(t))
>> axis square
>> title('(b)')
>> title('(a)')
>> subplot(1,2,2);plot(sin(t),exp(t))
>> title('(b)')
```


Figure 3.1: A two-pane plot of (a) and (b)



3.1.5 figure, clf and cla

figure(h), where h is an integer, creates a new figure window, or makes figure h the current figure. Subsequent plots are drawn in the current figure. h is called the handle of the figure. Handle graphics is discussed further in a later section of this chapter.

clf clears the current figure window. It also resets all properties associated with the axes, such as the hold state and the axis state.

cla deletes all plots and text from the current axes, i.e. leaves only the x- and y-axes and their associated information.

3.1.6 Logarithmic plots

The command

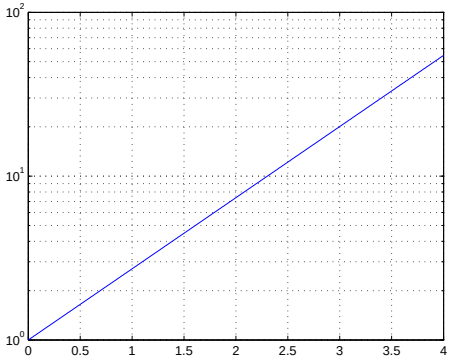
```
>> semilogy(x, y)
```

plots y with a $\log_{10}$ scale and x with a linear scale. For example, the statements

```
>> x = 0:.01:4;
semilogy(x, exp(x)), grid
```

produce the graph in Figure 3.2. Equal increments along the y-axis represent multiples of powers of 10. So, starting from the bottom, the grid lines are drawn at 1, 2, 3,..., 10, 20, 30 . . . , 100,.... Incidentally, the graph of e^x on these axes is a straight line, because the equation $y = e^x$ transforms into a linear equation when you take logs of both sides.

Figure 3.2: A logarithmic plot



3.1.7 Polar plots

The point (x, y) in cartesian coordinates is represented by the point (θ, r) in polar coordinates, where

$$x = rcos(\theta)$$
$$y = rsin(\theta)$$

and θ varies between 0 and 2π radians (360^o).
The command

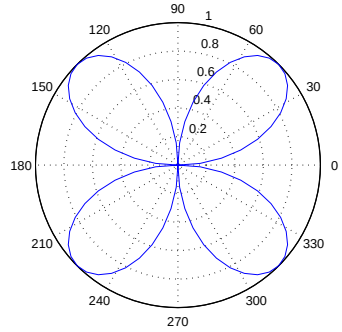
```
>> polar(theta, r)
```


generates a polar plot of the points with angles in theta and magnitudes in r.

As an example, the statements

```
>> x = 0:pi/40:2*pi;
>> polar(x, sin(2*x)),grid
```

Figure 3.3: Polar plot of θ against $sin(2\theta)$



3.2 3-D plots

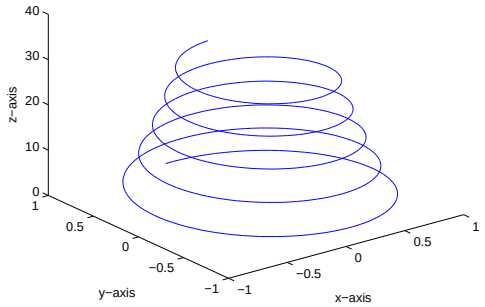
Matlab has a variety of functions for displaying and visualizing data in 3-D, either as lines in 3-D, or as various types of surfaces. This section provides a brief overview.
The function plot3 is the 3-D version of plot. The command

```
>> plot3(x, y, z)
```

draws a 2-D projection of a line in 3-D through the points whose coordinates are the elements of the vectors x, y and z. For example, the command

```
>>t = 0:pi/50:10*pi;
>>plot3(exp(-0.02*t).*sin(t), exp(-0.02*t).*cos(t),t)
>>xlabel('x-axis'), ylabel('y-axis'), zlabel('z-axis')
```

Figure 3.4: Examples of plot3



3.2.1 Animated 3-D plots with comet3

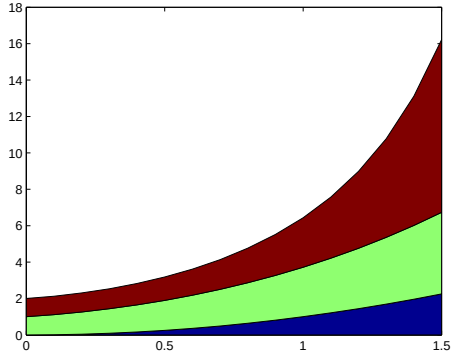
The function comet3 is similar to plot3 except that it draws with a moving 'comet head.' Use comet3 to animate the helix in Figure 3.4.

3.3 Other cool graphics functions

Here are more examples of interesting graphics functions. With each function a sample script and graph is shown. The list is by no means exhaustive and is meant to whet your appetite for more! You are encouraged to consult the list of graphics functions in the online **Matlab FUNCTION REFERENCE**. Each entry has excellent examples.

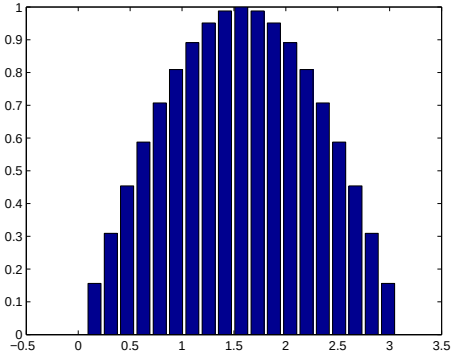
(a) **area**

```
>> x = 0:0.1:1.5;
>> area(x', [x.^2' exp(x)' exp(x.^2)'])
```

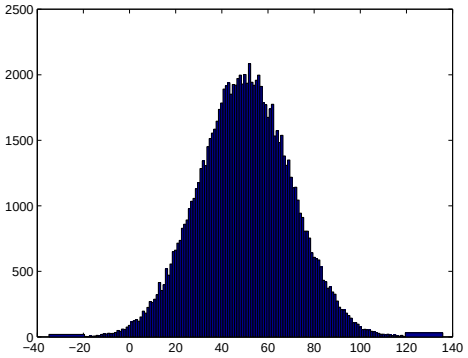
(b) **bar**

```
>>  x = 0:pi/20:pi;
>> bar(x,sin(x))
```



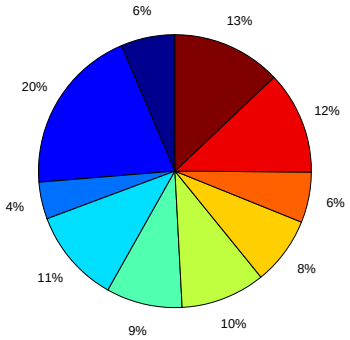
(c) **hist**

```
>> x = -20:120;
>> y = 50+20*randn(1,100000);
>> hist(y,x)
```



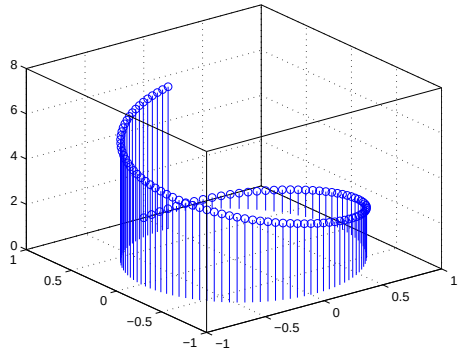
(d) **pie**

```
>> pie(rand(1,10))
```



(e) **stem3**

```
>> t = 0:pi/50:2*pi;
>> r = exp(-0.05*t);
>> stem3(r.*sin(t), r.*cos(t),t)
```

Chapter 4

POLYNOMIALS

A *polynomial* is a function of a single variable that can be expressed in the following form:

$$p(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + ... + a_{n-1}x^1 + a_n,$$

where the variable is x and the coefficients of the polynomial are represented by the values $a_0, a_1, ...$ and so on. The *degree* of a polynomial is equal to the largest value used as an exponent.
The polynomial $x^4+2x^3-13x^2-14x+24$ is represented in `Matlab` by the array $[1, 2, -13, -14, 24]$, i.e., by the coefficients of the polynomial starting with the highest power and ending with the constant term. If any power is missing from the polynomial its coefficient must appear in the array as a zero. Here are some of the things Matlab can do with polynomials.

4.1 Roots of a Polynomial

We can extract the roots of a polynomial. To calculate the roots of a polynomial given the coefficients, enter the coefficients in an array in descending order. Be sure to include zeroes where appropriate. The following command will find the roots of a polynomial:

```
>> p=[1 2 -13 -14 24];
```



```
>> r=roots(p)

r =

   -4.0000
    3.0000
   -2.0000
    1.0000
```

Note: The coefficients could be entered directly in the roots command. The same answer as above would be obtained using the following expression.

```
>> > r = roots([1 2 -13 -14 24])

r =

   -4.0000
    3.0000
   -2.0000
    1.0000
```

4.2 Find the polynomial from the roots

If you know that the roots of a polynomial are 1, 2, and 3, then you can find the polynomial in `Matlab's` array form this way

```
>> r=[1,2,3];
>> p=poly(r)

p =
```

```
     1     -6     11     -6
```

4.3 Multiply Polynomials

The command conv multiplies two polynomial coefficient arrays and returns the coefficient array of their product.

```
>> a=[1,0,1];
>> b=[1,0,-1];
>> c=conv(a,b)

c =

     1     0     0     0    -1
```

4.4 Divide Polynomials

Matlab can do it with the commanddeconv, giving you the quotient and the remainder.

```
>> a=[1,1,1];    % a=x^2+x+1
>> b=[1,1];      % b=x+1
% now divide b into a finding the quotient and remainder
>> [q,r]=deconv(a,b)

q =

     1     0


r =
```



```
     0     0     1
```

This means that $q = x + 0 = x$ and $r = 0x^2 + 0x + 1 = 1$, so

$$\frac{x^2 + x + 1}{x + 1} = x + \frac{1}{x + 1}.$$

4.5 First Derivative

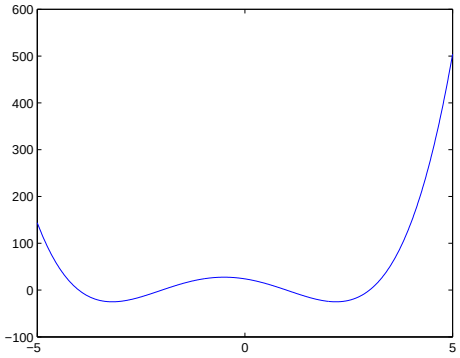
Matlab can take a polynomial array and return the polynomial array of its derivative:

```
>> a=[1,1,1,1];   % x^3 + x^2 + x + 1
>> ap=polyder(a)

ap =

     3     2     1
```

4.6 Evaluate a Polynomial

If you have an array of x-values and you want to evaluate a polynomial at each one, you can use the command ployval(p,x) as follows:

```
% define the polynomial
>> a=[1 2 -13 -14 24];
% load the x-values
>> x=-5:.01:5;
% evaluate the polynomial
>> y=polyval(a,x);
% plot it
>> plot(x,y)
```



4.7 Fitting Data to a Polynomial

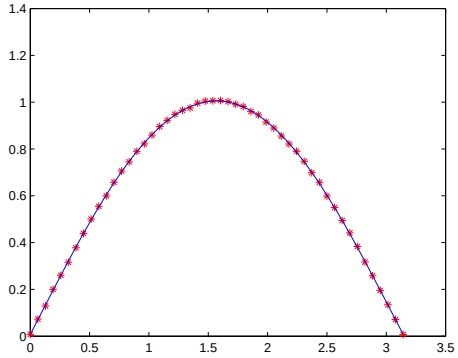
If you have some data in the form of arrays (x, y), Matlab will do a least-squares fit of a polynomial of any order you choose to this data i.e, to determining the coefficients of a polynomial that is the best fit of a given data, you can use the Matlab commandployfit.

The structure of the command is polyfit(x, y, n), where x, y are the data vectors and n is the order of the polynomial for which the least-squares fit is desired. In the next example we will let the data be the sine function between 0 and π and we will fit a polynomial of order 4 to it. Then we will plot the two functions on the same frame to see if the fit is any good.

```
>> x=linspace(0,pi,50);
% make a sine function with 1% random error on it
>> f=sin(x)+.01*rand(1,length(x));
% fit to the data
>> coeff=polyfit(x,f,4)

coeff =
```



```
    0.0379   -0.2374    0.0591    0.9813    0.0074
% evaluate the fit
>> g=polyval(coeff,x);
% plot fit and data together
>> plot(x,f,'r*',x,g,'b-')
```



`Sample Problem 4.1`. Fit the following data describing the accumulation of species A over time to a second order polynomial, then by using this polynomial, predict the accumulation at 15 hours.

Time(hr)	1	3	5	7	8	10
Mass A acc.	9	55	141	267	345	531

Solution:

First, input the data into vectors, let:

```
>> a = [9, 55, 141, 267, 345, 531];
>> time = [1, 3, 5, 7, 8, 10];
```

Now fit the data using polyfit

```
>> coeff = polyfit(time,a,2)
```

```
coeff =

    5.0000    3.0000    1.0000
```

So, Mass $A = 5 \times (time)^2 + 3 \times (time) + 1$
Therefore to calculate the mass A at 15 hours

```
>> MApred = polyval(coeff,15)

MApred =

   1.1710e+03
```

`Sample Problem 4.2.` Use `Matlab` to fit the following vapor pressure vs temperature data in fourth order polynomial. Then calculate the vapor pressure when $T = 100^\circ$ C.

Temp (C)	-36.7	-19.6	-11.5	-2.6	7.6	15.4	26.1	42.2	60.6	80.1
Pre. (kPa)	1	5	10	20	40	60	100	200	400	760

Solution:

```
>> vp = [ 1, 5, 10, 20, 40, 60, 100, 200, 400, 760];
>> T = [-36.7, -19.6, -11.5, -2.6, 7.6, 15.4, 26.1, 42.2, 60.6, 80.1];
>> p=polyfit(T,vp,4)
%evaluate the polynomial at T=100
pre= polyval(p,100)
p =

    0.0000    0.0004    0.0360    1.6062   24.6788
```



```
>> pre= polyval(p,100)

pre =

   1.3552e+03
```

Chapter 5

Programming in Matlab

5.1 Script Files

A *script* is a sequence of ordinary statements and functions used at the command prompt level. A script is invoked the command prompt level by typing the file-name or by using the pull down menu. Scripts can also invoke other scripts.

The commands in the Command Window cannot be saved and executed again. Also, the Command Window is not interactive. To overcome these difficulties, the procedure is first to create a file with a list of commands, save it and then run the file. In this way, the commands contained are executed in the order they are listed when the file is run. In addition, as the need arises, one can change or modify the commands in the file; the file can be saved and run again. The files that are used in this fashion are known as script files. Thus, a script file is a text file that contains a sequence of Matlab commands. *Script file* can be edited (corrected and/or changed) and executed many times.

5.1.1 Creating and Saving a Script File

Any text editor can be used to create script files. In Matlab, script files are created and edited in the Editor/ Debugger Window. This window can be

opened from the Command Window. From the Command Window, select *File*, *New* and then M-file. Once the window is open, the commands of the script file are typed line by line. The commands can also be typed in any text editor or word processor program and then copied and pasted in the Editor/Debugger Window. The second type of M-files is the *function file*. Function file enables the user to extend the basic library functions by adding ones own computational procedures. Function M-files are expected to return one or more results. Script files and function files may include reference to other Matlab toolbox routines.

Matlab function file begins with a header statement of the form:

```
function (name of result or results) = name (argument list)
```

Before a script file can be executed it must be saved. All script files must be saved with the extension ".m". Matlab refers to them as M-files. When using Matlab M-files editor, the files will automatically be saved with a ".m" extension. If any other text editor is used, the file must be saved with the ".m" extension, or Matlab will not be able to find and run the script file. This is done by choosing *Save As* from the *File* menu, selecting a location, and entering a name for the file. The names of user defined variables, predefined variables, Matlab commands or functions should not be used to name script files.

5.1.2 Running a Script File

A script file can be executed either by typing its name in the Command Window and then pressing the *Enter* key, directly from the Editor Window by clicking on the *Run* icon. The file is assumed to be in the current directory, or in the search path.

5.1.3 Input to a Script File

There are three ways of assigning a value to a variable in a script file.

1- The variable is defined and assigned value in the script file.

2- The variable is defined and assigned value in the Command Window.

3- The variable is defined in the script file, but a specified value is entered in the Command Window when the script file is executed.

5.2 Comparison Between Script Flies and Function Files

understanding exactly the differences between script and function files since, for many of the problems that they are asked to solve using `Matlab`, either type of file can be used. The similarities and differences between script and function files are summarized below.

- Both script and function files are saved with the extension .m (that is why they are sometimes called M-files).
- The first executable line in a function file is (must be) the function definition line.
- The variables in a function file are local. The variables in a script file are recognized in the Command Window.
- Script files can use variables that have been defined in the workspace.
- Script files contain a sequence of `Matlab` commands (statements).
- Function files can accept data through input arguments and can return data through output arguments.
- When a function file is saved, the name of the file should be the same as the name of the function.

5.3 Anonymous and Inline Functions

5.3.1 Anonymous Functions

An anonymous function is a simple (one-line) user-defined function that is defined without creating a separate function file (M-file). Anonymous functions can be constructed in the Command Window, within a script file, or inside a regular user-defined function.

An anonymous function is created by typing the following command:

name = @ (arglist) expr

where name refers to the name of the anonymous function, @ is the symbol @, arglist a list of input arguments (independent variables) and expr is the Mathematical expression.
A simple example is: cube = $@(x)x^3$, which calculates the cube of the input argument.
Once an anonymous function is defined, it can be used by typing its name and a value for the argument (or arguments) in parentheses (see examples that follow).

Example of an anonymous function with one independent variable:
The function $f(x) = \frac{e^{x^2}}{\sqrt{x^2+5}}$ can be defined (in the Command Window) as an anonymous function for x as a scalar by:

```
>> FA = @ (x) exp(x^2)/sqrt(x^2+5)

FA =

    @(x)exp(x^2)/sqrt(x^2+5)
```

If a semicolon is not typed at the end, `Matlab` responds by displaying the

function. The function can then be used for different values of x, as shown below.

```
>> FA(2)

ans =

   18.1994

>> z = FA(3)

z =

   2.1656e+03
```

If x is expected to be an array, with the function calculated for each element, then the function must be modified for element-by-element calculations.

```
>> FA = @ (x) exp(x.^2)./sqrt(x.^2+5)

FA =

    @(x)exp(x.^2)./sqrt(x.^2+5)

>> FA([1 0.5 2])

ans =

    1.1097    0.5604   18.1994
```

Example of an anonymous function with several independent variables:

The function $f(x, y) = 2x^2 - 4xy + y^2$ can be defined as an anonymous function by:

```
>> HA = @ (x,y) 2*x^2 - 4*x*y + y^2

HA =

    @(x,y)2*x^2-4*x*y+y^2
```

Then the anonymous function can be used for different values of x and y. For example, typing HA(2,3) gives:

```
>> HA(2,3)

ans =

    -7
```

5.3.2 Inline Functions

Similar to an anonymous function, an inline function is a simple user-defined function that is defined without creating a separate function file (M-file). As already mentioned, anonymous functions replace the inline functions used in earlier versions of Matlab. Inline functions are created with the inline command according to the following format:

name = inline('math expression typed as a string')

A simple example is cube = inline('x^3'), which calculates the cube of the input argument.
Once the function is defined it can be used by typing its name and a value for the argument (or arguments) in parentheses (see example below). For

example, the function: $f(x) = \frac{e^{x^2}}{\sqrt{x^2+5}}$ can be defined as an inline function for x by:

```
>> FA=inline('exp(x.^2)./sqrt(x.^2+5)')

FA =

     Inline function:
     FA(x) = exp(x.^2)./sqrt(x.^2+5)

>> FA(2)

ans =

   18.1994

>> FA([1 0.5 2])

ans =

    1.1097    0.5604   18.1994
```

An inline function that has two or more independent variables can be written by using the following format:

name = inline('mathematical expression','arg1','arg2','arg3')

In the format shown here the order of the arguments to be used when calling the function is defined. If the independent variables are not listed in the command, Matlab arranges the arguments in alphabetical order. For example, the function $f(x, y) = 2x^2 - 4xy + y^2$ can be defined as an inline function by


```
>> HA=inline('2*x^2-4*x*y+y^2')

HA =

     Inline function:
     A(x,y) = 2*x^2-4*x*y+y^2

>> HA(2,3)

ans =

    -7

>> HA=inline('2*x^2-4*x*y+y^2','x','y')

HA =

     Inline function:
     HA(x,y) = 2*x^2-4*x*y+y^2

>> HA(2,3)

ans =

    -7
>> HA=inline('2*x^2-4*x*y+y^2','y','x')  % we use y first

HA =
```

```
     Inline function:
     HA(y,x) = 2*x^2-4*x*y+y^2

>> HA(2,3)    % since we use y first, it means that y = 2 and x = 3

ans =

    -2
```

5.4 The feval Command

The feval (short for "function evaluate") command evaluates the value of a function for a given value (or values) of the function's argument (or arguments). The format of the command is

variable = feval('function name', argument value)

or

variable = feval(@function name, argument value)

The value that is determined by feval can be assigned to a variable, or if the command is typed without an assignment, Matlab displays ans = and the value of the function.

- The function can be a built-in or a user-defined function.
- If there is more than one input argument, the arguments are separated with commas.
- If there is more than one output argument, the variables on the left-hand side of the assignment operator are typed inside brackets and separated with commas.


```
>> feval('sqrt',64)

ans =

     8

>> feval(@sqrt,64)

ans =

     8
```

5.5 Relational and Logical Operators

A relational operator compares two numbers by finding whether a comparison statement is true or false. A logical operator examines true/false statements and produces a result which is true or false according to the specific operator. Relational and logical operators are used in mathematical expressions and also in combination with other commands to make decision that control the flow of a computer program.

Matlab has six relational operators as shown in Table 5.1. Examples:

Table 5.1: Relational operators.

Relational operator	Interpretation
$<$	Less than
$\leq$	Less than or equal
$>$	Greater than
$\geq$	Greater than or equal
==	Equal
~ =	Not equal

```
>> A=1:9, B=8-A
```

```
A =

     1     2     3     4     5     6     7     8     9


B =

     7     6     5     4     3     2     1     0    -1

>> tf1 = A <=3

tf1 =

     1     1     1     0     0     0     0     0     0

>> tf2 = A > B

tf2 =

     0     0     0     0     1     1     1     1     1

>> tf3 = (A==B)

tf3 =

     0     0     0     1     0     0     0     0     0

>> tf4 = B-(A>2)
```



```
tf4 =

     7     6     4     3     2     1     0    -1    -2
```

- tf1 finds elements of A that are less than or equal to 3. Ones appear in the result where $A \leq 3$ and zeroes appear where $A > 3$.
- tf2 finds elements of A that are greater than those in B.
- tf3 finds elements of A that are equal to those in B.
- tf4 finds where $A > 2$ and subtracts the resulting vector from B. This shows that since the output of logical operations are numerical arrays of ones and zeros, they can be used in mathematical operations.

Note that = and == mean two different things: == compares two variables and returns ones where they are equal and zeros where they are not; on the other hand, = is used to assign the output of an operation to a variable.
The logical operators in Matlab are shown in Table 5.2. A fourth logical

Table 5.2: Logical operators.

Logical Operator	Description
&	And
|	Or
∼	Not

operator is implemented as a function:

- xor(A,B) Exclusive or: Returns ones where either A or B is True (nonzero); returns False (zero) where both A and B are False (zero) or both are True (nonzero).

Table 5.3: Logical operators.

A	B	$\sim A$	$A \mid B$	$A\&B$	$xor(A,B)$
0	0	1	0	0	0
0	1	1	1	0	1
1	0	0	1	0	1
1	1	0	1	1	0

Definitions of the logical operators, with 0 representing False and 1 representing True.
Examples:

```
>> A=1:9

A =

     1     2     3     4     5     6     7     8     9

>> tf1 = A>3

tf1 =

     0     0     0     1     1     1     1     1     1

>> tf2 = ~(A>3)

tf2 =

     1     1     1     0     0     0     0     0     0

>> tf3 = (A>1)&(A<5)
```



```
tf3 =

     0     1     1     1     0     0     0     0     0

>> tf4 = xor((A>1),(A<5))

tf4 =

     1     0     0     0     1     1     1     1     1
```

5.6 Relational and Logical Functions

Matlab provides several useful relational and local functions that operate on scalars, vectors, and matrices. The following is a partial list of these functions.

Table 5.4: Built-in functions for handling arrays.

Function	Description	Example
all(A)	Returns 1 (true) if all elements in a vector A are true (non-zero). Returns 0 (false) if one or more elements are false (zero). If A is a matrix, treats columns 1's and 0's.	>> A = [5 3 11 7 8 15]; >> all(A) ans = 1 >> B = [3 6 11 4 0 13] >> all(B) ans = 0
any(A)	Returns 1 (true) if any element in a vector A is true (non-zero). Returns 0 (false) if all elements are false (zero). If A is a matrix, treats columns of A as vectors, returns a vector with 1's and 0's	>> A = [5 0 14 0 0 13]; >> any(A) ans = 1 >> B = [0 0 0 0 0 0]

Continued on next page

Table 5.4 – continued from previous page

Function	Description	Example
		>> any(B) ans = 0
find(A)	If A is a vector, returns the indices of the non-zero elements.	>> A = [0 7 4 2 8 0 0 3 9]; >> find(A) ans = 2 3 4 5 8 9
find(A > d)	If A is a vector, returns the address of the elements that are larger than d (any relational operator can be used).	>> find(A > 4) >> ans = 4 5 6

5.7 Flow Control

Matlab has four kinds of statements you can use to control the flow through your code:

if, else and elseif execute statements based on a logical test.
switch, case and otherwise execute groups of statements based on a logical test.
while and end execute statements an indefinite number of times, based on a logical test.
for and end execute statements a fixed number of times.

the if ··· end Structure

The basic form of an if statement is:

```
if test
   statements
end
```


The *test* is an expression that is either 1 (true) or 0 (false). The *statements* between the if and end statements are executed if the *test* is true. If the *test* is false the statements will be ignored and execution will resume at the line after the end statement. The test expression can be a vector or matrix, in which case all the elements must be equal to 1 for the statements to be executed. Further tests can be made using the elseif and else statements.

```
if a < 30
   count = count + 1
   disp a
end
```

The if ··· else Structure

This structure allows you to execute a set of statements if a logical condition is true and to execute a second set if the condition is false. Its general syntax is

```
if test
   statement_1
else
   statement_2
end
```

The if ··· elseif Structure

It often happens that the false option of an if ... else structure is another decision. This type of structure often occurs when we have more than two options for a particular problem setting. For such cases, a special form of decision structure, the if ... elseif has been developed. It has the general syntax:

```
if test_1
   statement_1
elseif test_2
   statement_2
   .
   .
   .
else
   statement_else
end
```

`Problem Statement`. For a scalar, the built-in `Matlab` sign function returns the sign of its argument (-1, 0, 1). Here's a `Matlab` session that illustrates how it works:

```
>> sign(25.6)

ans =

     1

>> sign(-0.776)

ans =

    -1

>> sign(0)

ans =

     0
```


Develop an M-file to perform the same function. `Solution`. First, an if structure can be used to return 1 if the argument is positive:

```
function sgn = mysign(x)
% mysign(x) returns 1 if x is greater than zero.
if x > 0
sgn = 1;
end
```

This function can be run as

```
function sgn = mysign(x)
>> mysign(25.6)

ans =

     1
```

Although the function handles positive numbers correctly, if it is run with a negative or zero argument, nothing is displayed. To partially remedy this shortcoming, an if... else structure can be used to display 1 if the condition is false:

```
function sgn = mysign(x)
% mysign(x) returns 1 if x is greater than zero.
% -1 if x is less than or equal to zero.
if x > 0
sgn = 1;
else
sgn = -1;
end
```

This function can be run as

```
>> mysign(-0.776)

ans =

     -1
```

Although the positive and negative cases are now handled properly, -1 is erroneously returned if a zero argument is used. An if ... elseif structure can be used to incorporate this final case:

```
function sgn = mysign(x)
% mysign(x) returns 1 if x is greater than zero.
% -1 if x is less than zero.
% 0 if x is equal to zero.
if x > 0
sgn = 1;
elseif x < 0
sgn = -1;
else
sgn = 0;
end
```

The function now handles all possible cases. For example,

```
>> mysign(0)

ans =

     0
```


5.7.1 The switch Structure

The basic form of a switch statement is:

```
    switch test
           case result1
                statements
           case result2
                statements
                .
                .
                .
           otherwise
                statements
    end
```

The respective statements are executed if the value of test is equal to the respective results. If none of the cases are true, the otherwise statements are done. Only the first matching case is carried out. As an example, here is function that displays a message depending on the value of the string variable, grade.

```
grade = 'B';
switch grade
  case 'A'
   disp('Excellent')
  case 'B'
   disp('Good')
  case 'C'
   disp('Mediocre')
  case 'D'
   disp('Whoops')
```

```
  case 'F'
   disp('Would like fries with your order?')
  otherwise
   disp('Huh!')
end
```

When this code was executed, the message "Good" would be displayed.

5.7.2 Loops

As the name implies, loops perform operations repetitively. There are two types of loops, depending on how the repetitions are terminated. A for loop ends after a specified number of repetitions. A while loop ends on the basis of a logical condition.

The for ... end Structure

A for loop repeats statements a specific number of times. Its general syntax is

```
for index = start:increment:stop
    statements
end
```

The for loop operates as follows. The index is a variable that is set at an initial value, start. The program then compares the index with a desired final value, stop. If the index is less than or equal to the stop, the program executes the statements. When it reaches the end line that marks the end of the loop, the index variable is increased by the increment and the program loops back up to the for statement. The process continues until the index becomes greater than the stop value. At this point, the loop terminates as the

Note that if an increment of 1 is desired (as is often the case), the increment can be dropped.
For example,

```
for i = 1:5
    disp(i);
end
```

When this executes, Matlab would display in succession, 1, 2, 3, 4, 5. In other words, the default increment is 1.

The size of the increment can be changed from the default of 1 to any other numeric value. It does not have to be an integer, nor does it have to be positive. For example, step sizes of 0.2, 1, or 5, are all acceptable.

If a negative increment is used, the loop will "countdown" in reverse. For such cases, the loop's logic is reversed. Thus, the stop is less than the start and the loop terminates when the index is less than the stop.

For example,

```
for j = 10:-1:1
    disp(j);
end
```

When this executes, Matlab would display the classic "countdown" sequence: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.

`Problem Statement`. Develop an M-file to compute the factorial

$$0! = 0$$
$$1! = 1$$
$$2! = 2 \times 1 = 2$$
$$3! = 3 \times 2 \times 1 = 6$$
$$4! = 4 \times 3 \times 2 \times 1 = 24$$
$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$
$$\vdots$$

`Solution`. A simple function to implement this calculation can be developed as

```
function fout = factor(n)
% factor(n):
% Computes the product of all the integers from 1 to n.
x = 1;
for i = 1:n
x = x * i;
end
fout = x;
end
```

which can be run as

```
>> factor(5)

ans =

    125
```

This loop will execute 5 times (from 1 to 5). At the end of the process, x will hold a value of 5! (meaning 5 factorial or $1 \times 2 \times 3 \times 4 \times 5 = 120$).

Notice what happens if $n = 0$. For this case, the for loop would not execute, and we would get the desired result, $0! = 1$.

Vectorization

The for loop is easy to implement and understand. However, for Matlab, it is not necessarily the most efficient means to repeat statements a specific number of times. Because of Matlab's ability to operate directly on arrays, vectorization provides a much more efficient option. For example, the following for loop structure:

```
i = 0;
for t = 0:0.02:50
  i = i + 1;
  y(i) = cos(t);
end
```

can be represented in vectorized form as

```
t = 0:0.02:50;
y = cos(t);
end
```

It should be noted that for more complex code, it may not be obvious how to vectorize the code. That said, wherever possible, vectorization is recommended.

Preallocation of Memory

Matlab automatically increases the size of arrays every time you add a new element. This can become time consuming when you perform actions such as adding new values one at a time within a loop. For example, here is some code that sets value of elements of y depending on whether or not values of t are greater than one:

```
t = 0:.01:5;
for i = 1:length(t)
  if t(i)>1
    y(i) = 1/t(i);
  else
    y(i) = 1;
 end
end
```

For this case, Matlab must resize y every time a new value is determined. The following code preallocates the proper amount of memory by using a vectorized statement to assign ones to y prior to entering the loop.

```
t = 0:.01:5;
y = ones(size(t));
for i = 1:length(t)
  if t(i)>1
    y(i) = 1/t(i);
  else
    y(i) = 1;
 end
end
```

Thus, the array is only sized once. In addition, preallocation helps reduce memory fragmentation, which also enhances efficiency.

The while Structure

A while loop repeats as long as a logical condition is true. Its general syntax is

```
while test
   statements
```



```
end
```

The statements between the while and the end are repeated as long as the test is true. A simple example is

```
x = 8
while x > 0
  x = x - 3;
  disp(x)
end
```

When this code is run, the result is

```
x =
   8
   5
   2
  -1
```

A quick way to 'comment out' a slab of code in an m-file is to enclose it between a while 0 and end statements. The enclosed code will never be executed.

The while ... break Structure

Although the while structure is extremely useful, the fact that it always exits at the beginning of the structure on a false result is somewhat constraining. For this reason, languages such as Fortran 90 and Visual Basic have special structures that allow loop termination on a true test anywhere in the loop. Although such structures are currently not available in Matlab, their functionality can be mimicked by a special version of the while loop. The syntax of this version, called a while ... break structure, can be written as

```
while (1)
  statements
  if test, break, end
  statements
end
```

where break terminates execution of the loop. Thus, a single line if is used to exit the loop if the tests true. Note that as shown, the break can be placed in the middle of the loop (i.e., with statements before and after it). Such a structure is called a *midtest loop*.

If the problem required it, we could place the break at the very beginning to create a *pretest loop*. An example is

```
while (1)
  if x < 0, break, end
  x = x - 5;
end
```

Notice how 5 is subtracted from x on each iteration. This represents a mechanism so that the loop eventually terminates. Every decision loop must have such a mechanism. Otherwise it would become a so-called *infinite loop* that would never stop.

Alternatively, we could also place the if ... break statement at the very end and create a *posttest loop*,

```
while (1)
  x = x - 5;
  if x < 0, break, end
end
```

It should be clear that, in fact, all three structures are really the same. That is, depending on where we put the exit (beginning, middle, or end) dictates whether we have a pre-, mid- or posttest. It is this simplicity that

led the computer scientists who developed Fortran 90 and Visual Basic to favor this structure over other forms of the decision loop such as the conventional while structure

The pause Command

There are often times when you might want a program to temporarily halt. The command pause causes a procedure to stop and wait until any key is hit. A nice example involves creating a sequence of plots that a user might want to leisurely peruse before moving on to the next. The following code employs a for loop to create a sequence of interesting plots that can be viewed in this manner:

```
for n = 3:10
  mesh(magic(n))
  pause
end
```

The pause can also be formulated as pause(n), in which case the procedure will halt for n seconds. This feature can be demonstrated by implementing it in conjunction with several other useful Matlab functions.

Examples

Problem Statement. The roots of a quadratic equation

$$f(x) = ax^2 + bx + c$$

can be determined with the quadratic formula

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

Develop a function to implement this formula given values of the coefficients.

Solution. *Top-down* design provides a nice approach for designing an algorithm to compute the roots. This involves developing the general

structure without details and then refining the algorithm. To start, we first recognize that depending on whether the parameter a is zero, we will either have "special" cases (e.g., single roots or trivial values) or conventional cases using the quadratic formula. This "big-picture" version can be programmed as

```
function quadroots(a, b, c)
% quadroots: roots of quadratic equation
% quadroots(a,b,c): real and complex roots
% of quadratic equation
% input:
% a = second-order coefficient
% b = first-order coefficient
% c = zero-order coefficient
% output:
% r1 = real part of first root
% i1 = imaginary part of first root
% r2 = real part of second root
% i2 = imaginary part of second root
if a == 0
   %special cases
else
   %quadratic formula
end
```

Next, we develop refined code to handle the "special" cases:

```
%special cases
if b ~= 0
   %single root
   r1 = -c / b
```



```
else
   %trivial solution
   disp('Trivial solution. Try again')
end
```

And we can develop refined code to handle the quadratic formula cases:

```
%quadratic formula
d = b ^ 2 - 4 * a * c;
if d >= 0
   %real roots
   r1 = (-b + sqrt(d)) / (2 * a)
   r2 = (-b - sqrt(d)) / (2 * a)
else
   %complex roots
   r1 = -b / (2 * a)
   i1 = sqrt(abs(d)) / (2 * a)
   r2 = r1
   i2 = -i1
end
```

We can then merely substitute these blocks back into the simple "big-picture" framework to give the final result:

```
function quadroots(a, b, c)
% quadroots: roots of quadratic equation
% quadroots(a,b,c): real and complex roots
% of quadratic equation
% input:
% a = second-order coefficient
% b = first-order coefficient
% c = zero-order coefficient
```

```
% output:
% r1 = real part of first root
% i1 = imaginary part of first root
% r2 = real part of second root
% i2 = imaginary part of second root
if a == 0
  %special cases
  if b ~= 0
    %single root
    r1 = -c / b
  else
    %trivial solution
    disp('Trivial solution. Try again')
  end
else
  %quadratic formula
  d = b ^ 2 - 4 * a * c;
  if d >= 0
    %real roots
    r1 = (-b + sqrt(d)) / (2 * a)
    r2 = (-b - sqrt(d)) / (2 * a)
  else
    %complex roots
    r1 = -b / (2 * a)
    i1 = sqrt(abs(d)) / (2 * a)
    r2 = r1
    i2 = -i1
  end
end
```


As highlighted by the shading, notice how indentation helps to make the underlying logical structure clear. Also notice how .modular. the structures are. Here is a command window session illustrating how the function performs:

```
>> quadroots(1,1,1)
r1 =
   -0.5000
i1 =
   0.8660
r2 =
   -0.5000
i2 =
   -0.8660
>> quadroots(1,5,1)
r1 =
   -0.2087
r2 =
   -4.7913
>> quadroots(0,5,1)
r1 =
   -0.2000
>> quadroots(0,0,0)
Trivial solution. Try again
```

`Problem Statement`. Develop an M-file function to determine the average value of a function over a range. Illustrate its use for the bungee jumper velocity over the range from $t = 0$ to $12s$:

$$v(t) = \sqrt{\frac{gm}{c_d}} \tanh\left(\sqrt{\frac{gc_d}{m}} t\right)$$

where $g = 9.81$, $m = 68.1$, and $c_d = 0.25$.

Solution. The average value of the function can be computed with standard Matlab commands as

```
>> t=linspace(0,12);
>> v=sqrt(9.81*68.1/0.25)*tanh(sqrt(9.81*0.25/68.1)*t);
>> mean(v)

ans =

   36.0870
```

We can write an M-file to perform the same computation:

```
function favg = funcavg(a,b,n)
% funcavg: average function height
% favg=funcavg(a,b,n): computes average value
% of function over a range
% input:
% a = lower bound of range
% b = upper bound of range
% n = number of intervals
% output:
% favg = average value of function
x = linspace(a,b,n);
y = func(x);
favg = mean(y);
end
function f = func(t)
f=sqrt(9.81*68.1/0.25)*tanh(sqrt(9.81*0.25/68.1)*t);
end
```


The main function first uses linspace to generate equally spaced x values across the range. These values are then passed to a subfunction $func$ in order to generate the corresponding y values. Finally, the average value is computed. The function can be run from the command window as

```
>> funcavg (0,12,60)

ans =

    36.0127
```

Chapter 6

Matlab Applications

Since this is an introductory course, the applications are not extensive. They are illustrative. You ought to recognize that the kinds of problems you actually can solve with Matlab are much more challenging than the examples provided.

Sample Problem 6.1: **Exponential growth and decay**

A model for exponential growth or decay of a quantity is given by

$$A(t) = A_0 e^{kt},$$

where $A(t)$ and A_0 are the quantity at time t and time 0, respectively, and k is a constant unique to the specific application. For function name and arguments use $\mathsf{At} = \mathsf{expGD(A0,At1,t1,t)}$, where the output argument At corresponds to, and for input arguments use $\mathsf{A0}$, $\mathsf{At1}$, $\mathsf{t1}$, t, corresponding to $\mathsf{A_0}$, $\mathsf{A(t_1)}$, $\mathsf{t_1}$, and t, respectively. Write a user-defined function that uses this model to predict the quantity at time t from knowledge of and at some other time.

Use the function file in the Command Window for the following two cases:

(a) The population of Mexico was 67 million in the year 1980 and 79 million in 1986. Estimate the population in 2000.

(b) The half-life of a radioactive material is 5.8 years. How much of a 7-gram sample will be left after 30 years?

Solution

To use the exponential growth model, the value of the constant k has to be determined first by solving for k in terms of $\mathsf{A_0}$, $\mathsf{A(t_1)}$, and $\mathsf{t_1}$,:

$$k = \frac{1}{t_1} \ln \frac{A(t_1)}{A_0}.$$

Once k is known, the model can be used to estimate the population at any time. The user-defined function that solves the problem is:

```
function At=expGD(A0,At1,t1,t)
% expGD calculates exponential growth and decay
% Input arguments are:
% A0: Quantity at time zero.
% At1: Quantity at time t1.
% t1: The time t1.
% t: time t.
% Output argument is:
% At: Quantity at time t.
k=log(At1/A0)/t1;  % Determination of k.
At=A0*exp(k*t);    % Determination of A(t).
```

Once the function is saved, it is used in the Command Window to solve the two cases.

For case a) $A_0 = 67$, $A(t_1) = 79$, $t_1 = 6$, and $t = 20$:

```
>> expGD(67,79,6,20)

ans =

    116.03   % Estimation of the population in the year 2000.
```

For case b) $A_0 = 7$, $A(t_1) = 3.5$, (since t_1 corresponds to the half-life, which is the time required for the material to decay to half of its initial quantity), $t_1 = 5.8$, and $t = 30$.

```
>> expGD(7,3.5,5.8,30)

ans =

    0.19      % The amount of material after 30 years.
```

Sample Problem 6.2: **Motion of a projectile**

Create a function file that calculates the trajectory of a projectile. The inputs to the function are the initial velocity and the angle at which the projectile is fired. The outputs from the function are the maximum height and distance. In addition, the function generates a plot of the trajectory. Use the function to calculate the trajectory of a projectile that is fired at a velocity of $230m/s$ at an angle of 39°.

Solution

The motion of a projectile can be analyzed by considering the horizontal and vertical components. The initial velocity v_0 can be resolved into horizontal and vertical components

$$v_{0x} = v_0 \cos(\theta), \quad and \quad v_{0y} = v_0 \sin(\theta).$$

In the vertical direction the velocity and position of the projectile are given by:

$$v_y = v_{0y} - gt, \quad and \quad y = v_{0y}t - \frac{1}{2}gt^2.$$

The time it takes the projectile to reach the highest point ($v_y = 0$) and the corresponding height are given by:

$$t_{hmax} = \frac{v_{0y}}{g}, \quad and \quad h_{max} = \frac{v_{0y}^2}{2g}.$$

The total flying time is twice the time it takes the projectile to reach the highest point, $t_{tot} = 2t_{hmax}$. In the horizontal direction the velocity is constant, and the position of the projectile is given by:

$$x = v_{0x}t.$$

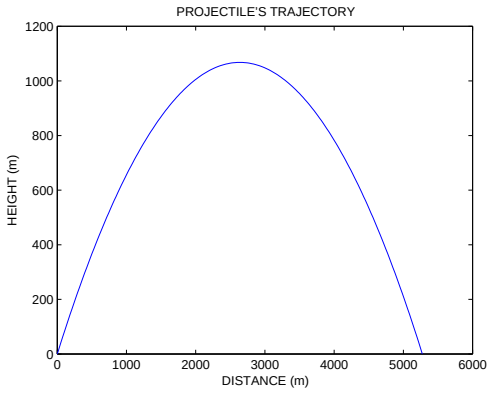
In Matlab notation the function name and arguments are entered as [hmax, dmax] = trajectory(v0, theta). The function file is:

```
function [hmax,dmax]=trajectory(v0,theta)
%trajectory calculates the max height and distance of a
%projectile, and makes a plot of the trajectory.
%
% Input arguments are:
% v0: initial velocity in (m/s).
% theta: angle in degrees.
% Output arguments are:
% hmax: maximum height in (m).
% dmax: maximum distance in (m).
% The function creates also a plot of the trajectory.
g=9.81;
v0x=v0*cos(theta*pi/180);
v0y=v0*sin(theta*pi/180);
thmax=v0y/g;
hmax=v0y^2/(2*g);
ttot=2*thmax;
dmax=v0x*ttot;
% Creating a trajectory plot
% Creating a time vector with 200 elements.
tplot=linspace(0,ttot,200);
x=v0x*tplot;
% Note the element-by-element multiplication.
y=v0y*tplot-0.5*g*tplot.^2;
plot(x,y)
xlabel('DISTANCE (m)')
ylabel('HEIGHT (m)')
```

```
title('PROJECTILE''S TRAJECTORY')
```

After the function is saved, it is used in the Command Window for a projectile that is fired at a velocity of $230m/s$ and an angle of 39°.

```
>> [hmax,dmax]=trajectory(230,39)

hmax =

   1.0678e+03


dmax =

   5.2746e+03
```



`Sample Problem 6.3`: Write a program to calculate average density, conductivity and specific heat for water in the range of temperatures from 0 to 50°C. Knowing that this parameters for water are a function of temperature such as the following equations.

The Density

$$\rho = 1200.92 - 1.0056T_K{}^\circ + 0.001084(T_K{}^\circ)^2.$$

The conductivity

$$K_c = 0.34 + 9.278 \times 10^{-4}T_K{}^\circ.$$

The Specific heat

$$C_P = 0.015539(T_K{}^\circ - 308.2)^2 + 4180.9.$$

Note: take 11 point of temperatures

Solution

```
function [Average_density,Average_conductivity,...
          Average_specificheat] = water()
ap=0;
aKc=0;
aCp=0;
% We note that the temperature in the above equations is given
% in Kelvin. Hence we must convert it from degree Celsius to
% Kelvin by the formula:
% " kelvin = degree Celsius + 273.15".
% The temperatures becomes now from 273 to 323 K.
% Also, note that the increment = 5 because we need 11 point
% of temperatures.
for T=273:5:323
p= 1200.92 - 1.0056*T+ 0.001084 * T^2;
Kc = 0.34 + 9.278 * 10^-4 *T;
Cp = 0.015539*(T - 308.2)^2 + 4180.9;
ap=ap+p;
aKc=aKc+Kc;
```

```
aCp=aCp+Cp;
end
Average_density=ap/11;
Average_conductivity=aKc/11;
Average_specificheat=aCp/11;
end
```

After the function is saved, it is used in the Command Window as follows:

```
>> [Average_density,Average_conductivity,...
    Average_specificheat] = water()

Average_density =

  997.7857


Average_conductivity =

    0.6165


Average_specificheat =
   4.1864e+03
```

Sample Problem 6.4: **Water flow in a river**

To estimate the amount of water that flows in a river during a year, a section of the river is made to have a rectangular cross section as shown. In the beginning of every month (starting at January 1st) the height h of the water and the speed v of the water flow are measured. The first day of measurement is taken as 1, and the last daywhich is January 1st of the next yearis day 366. The following data was measured:

Day	1	32	60	91	121	152	182	213	244	274	305	335	366
h (m)	2.0	2.1	2.3	2.4	3.0	2.9	2.7	2.6	2.5	2.3	2.2	2.1	2.0
v (m/s)	2.0	2.2	2.5	2.7	5.0	4.7	4.1	3.8	3.7	2.8	2.5	2.3	2.0

Use the data to calculate the flow rate, and then integrate the flow rate to obtain an estimate of the total amount of water that flows in the river during a year.

Solution

The flow rate, Q (volume of water per second), at each data point is obtained by multiplying the water speed by the width and height of the cross-sectional area of the water that flows in the channel:

$$Q = vwh \quad (m^3/h).$$

The total amount of water that flows is estimated by the integral:

$$V = (60 \cdot 60 \cdot 24) \int_{t_1}^{t_2} Q dt.$$

The flow rate is given in cubic meters per second, which means that time must have units of seconds. Since the data is given in terms of days, the integral is multiplied by $(60 \cdot 60 \cdot 24)$ s/day.

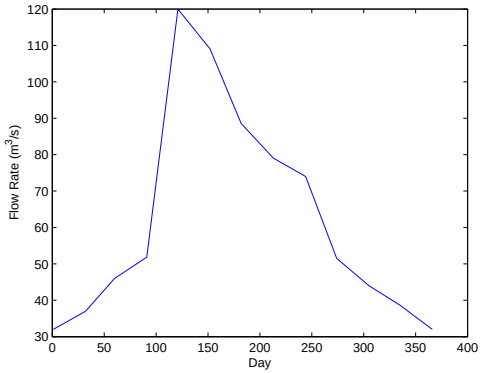
The following is a program written in a script file that first calculates Q and then carries out the integration using the trapz command. The program also generates a plot of the flow rate versus time.

```
>> w=8;
>> d=[1 32 60 91 121 152 182 213 244 274 305 335 366];
>> h=[2 2.1 2.3 2.4 3.0 2.9 2.7 2.6 2.5 2.3 2.2 2.1 2.0];
>> speed=[2 2.2 2.5 2.7 5 4.7 4.1 3.8 3.7 2.8 2.5 2.3 2];
>> Q=speed.*w.*h;
>> Vol=60*60*24*trapz(d,Q);
```

```
>> plot(d,Q)
>> xlabel('Day'), ylabel('Flow Rate (m^3/s)')
>> fprintf('The estimated amount of water that flows in the
river in a year is %g cubic meters.',Vol)
```

When we executed these steps in the Command Window, the estimated amount of water is displayed and the plot is generated. Both are shown below.

```
The estimated amount of water that flows in the river in
a year is 2.03095e+09 cubic meters.
```



`Sample Problem 6.5`: Make a table of (x vs. y) values for the three component ethanol, water and benzene a $T = 373.15$ k. Knowing that the vapor pressure of these three components are calculated by:
Ethanol $P_e^\circ = \exp(18.5242 - 3578.91/(T - 50.5))$.
Water $P_w^\circ = \exp(18.3036 - 3816.44/(T - 46.13))$.
Benzene $P_b^\circ = \exp(15.9008 - 2788.51/(T - 52.36))$.
Where

$$K_i = P_i^\circ / P_t, \quad P_t = 760, \quad y_i = K_i \times x_i.$$

Note: take 11 point for each component form 0 to 1

Solution

```
function A = E_W_B()
T=373.15;
Pe=exp(18.5242-3578.91/(T-50.5));
Pw=exp(18.3036-3816.44/(T-46.13));
Pb=exp(15.9008-2788.51/(T-52.36));
Ke=Pe/760;
Kw=Pw/760;
Kb=Pb/760;
Xe=0:.1:1; Xw = Xe; Xb = Xe;
Ye=Ke*Xe;
Yw=Kw*Xw;
Yb=Kb*Xb;
A = [Xe',Ye',Xw',Yw',Xb',Yb'];
end
```

After the function is saved, it is used in the Command Window as follows:

```
>> A = E_W_B()

A =

         0         0         0         0         0         0
    0.1000    0.2223    0.1000    0.1000    0.1000    0.1777
    0.2000    0.4445    0.2000    0.2000    0.2000    0.3554
    0.3000    0.6668    0.3000    0.3000    0.3000    0.5331
    0.4000    0.8890    0.4000    0.4000    0.4000    0.7107
    0.5000    1.1113    0.5000    0.5000    0.5000    0.8884
```

```
0.6000    1.3335    0.6000    0.6000    0.6000    1.0661
0.7000    1.5558    0.7000    0.6999    0.7000    1.2438
0.8000    1.7780    0.8000    0.7999    0.8000    1.4215
0.9000    2.0003    0.9000    0.8999    0.9000    1.5992
1.0000    2.2225    1.0000    0.9999    1.0000    1.7769
```

6.1 Solution to Differential Equations

6.1.1 Creating Symbolic Expressions

Symbolic expressions are mathematical expressions written in terms of symbolic variables. Once symbolic variables are created, they can be used for creating symbolic expressions. The symbolic expression is a symbolic object (the display is not indented). The form for creating a symbolic expression is:

Expression_name = Mathematical expression

A few examples are:

```
>> syms a b c x y   % Define a, b, c, x, and y as symbolic variables.
>> f=a*x^2+b*x + c  % Create the symbolic expression ax^{2} + bx + c
                    %and assign it to f.

f =

a*x^2 + b*x + c
```

Symbolic math functions can be used to solve a single equation, a system of equations and differential equations. For example:
solve(f): Solves a symbolic equation f for its symbolic variable. If f is a symbolic expression, this function solves the equation f = 0 for its symbolic variable.

solve(f1,...,fn): Solves the system of equations represented by f1,...,fn.

The symbolic function for solving ordinary differential equation is dsolve as shown below: dsolve('equation', 'condition'): Symbolically solves the ordinary differential equation specified by 'equation'. The optional argument 'condition' specifies a boundary or initial condition.

The symbolic equation uses the letter D to denote differentiation with respect to the independent variable. D followed by a digit denotes repeated differentiation. Thus, Dy represents dy/dx, and $D2y$ represents d^2y/dx^2.
For example, given the ordinary second order differential equation;

$$\frac{d^2x}{dt^2} + 5\frac{dx}{dt} + 3x = 7,$$

with the initial conditions $x(0) = 0$ and $\dot{x}(0) = 1$. The Matlab statement that determines the symbolic solution for the above differential equation is the following:

```
>> x = dsolve('D2x = 5*Dx  3*x + 7', 'x(0) = 0', 'Dx(0) =1')
```

The symbolic functions are summarized in Table 6.1

Table 6.1: Solution of equations

compose	Functional composition
dsolve	Solution of differential equations
finverse	Functional inverse
solve	Solution of algebraic equations

```
>> syms x y
>> f = x^2 +1

f =
```

```
x^2 + 1

>> g = y + 2

g =

y + 2

>> h = compose(f,g)

h =

(y + 2)^2 + 1
```

compose(f,g) returns f(g(y)) where f = f(x) and g = g(y). Here x and y are the symbolic variables of f and g respectively.

```
>> syms x
>> f = x^2

f =

x^2

>> finverse(f)

ans =

x^(1/2)
```



```
>> f = x^2 + 2*x +1

f =

x^2 + 2*x + 1

>> finverse(f)

ans =

x^(1/2) - 1
```

finverse(f) returns the functional inverse of f. Here f is an expression or function of one symbolic variable, for example, x. Then g is an expression or function, such that f(g(x)) = x.

```
>> syms x
>> solve(x^2 +2*x - 3)

ans =

  1
 -3

>> solve(x^3 + x + 2)

ans =

                    -1
 (7^(1/2)*i)/2 + 1/2
 1/2 - (7^(1/2)*i)/2
```

6.1.2 Calculus

There are four forms by which the symbolic derivative of a symbolic expression is obtained in Matlab. They are:
diff(f): Returns the derivative of the expression f with respect to the default independent variable.
diff(f,x): Returns the derivative of the expression f with respect to the variable x.
diff (f,n): Returns the nth derivative of the expression f with respect to the default independent variable.
diff(f,x,n): Returns the nth derivative of the expression f with respect to the variable x.

```
>> x = [1 3 5 7 9];
>> diff(x)

ans =

     2     2     2     2

>> syms x y
>> f = x^3*y + 5*sin(x) + 2

f =

5*sin(x) + x^3*y + 2

>> diff(f)

ans =
```



```
5*cos(x) + 3*x^2*y

>> diff(f,x)

ans =

5*cos(x) + 3*x^2*y

>> diff(f,y)

ans =

x^3

>> diff(f,2)

ans =

6*x*y - 5*sin(x)

>> diff(f,y,2)

ans =

0
```

The various forms that are used in Matlab to find the integral of a symbolic expression f are summarized as follows:
int(f): Returns the integral of the expression f with respect to the default independent variable.

int(f, x): Returns the integral of the expression f with respect to the variable x.

int(f,a,b): Returns the integral of the expression f with respect to the default independent variable evaluated over the interval [a, b], where a and b are numeric expressions.

int(f,x,a,b): Returns the integral of the expression f with respect to the variable x evaluated over the interval [a, b], where a and b are numeric expressions.

```
>> syms x y
>> f = x*cos(y) + 2

f =

x*cos(y) + 2

>> int(f)

ans =

(x*(x*cos(y) + 4))/2

>> int(f,0,pi)

ans =

(pi*(pi*cos(y) + 4))/2

>> int(f,y)
```



```
ans =

2*y + x*sin(y)

>> int(f,y,0,pi)

ans =

2*pi
```