

Introductory MATLAB

Core Topics

Starting with MATLAB (A.2).

Arrays (A.3).

Mathematical operations with arrays (A.4).

Script files (A.5).

Plotting (A.6).

User-defined functions and function files (A.7).

Anonymous functions (A.8).

Function functions (A.9).

Subfunctions (A.10)

Programming in MATLAB (A.11).

A.1 BACKGROUND

MATLAB is a powerful language for technical computing. The name MATLAB stands for MATrix LABoratory because its basic data element is a matrix (array). MATLAB can be used for mathematical computations, modeling and simulations, data analysis and processing, visualization and graphics, and algorithm development.

MATLAB is widely used in universities and colleges in introductory and advanced courses in mathematics, science, and especially in engineering. In industry the software is used in research, development, and design. The standard MATLAB program has tools (built-in functions) that can be used to solve common problems. In addition, MATLAB has optional toolboxes that are a collection of specialized programs designed to solve specific types of problems. Examples include toolboxes for signal processing, symbolic calculations, and control systems.

This appendix is a brief introduction to MATLAB. It presents the most basic features and syntax that will enable the reader to follow the use of MATLAB in this book. For a more complete introduction, the reader is referred to MATLAB: An Introduction with Applications, Fourthrd Edition, by Amos Gilat, Wiley, 2011.

A.2 STARTING WITH MATLAB

It is assumed that the software is installed on the computer and that the user can start the program. When the program is running, eight windows can be used. A list of the various windows and their purpose is

given in Table A-1. Four of the windows—the Command Window, the Figure Window, the Editor Window, and the Help Window—are the most commonly used.

Table A-1: MATLAB Windows

Window	Purpose
Command Window	Main window, enters variables, runs programs.
Figure Window	Contains output from graphics commands.
Editor Window	Creates and debugs script and function files.
Help Window	Provides help information.
Launch Pad Window	Provides access to tools, demos, and documentation.
Command History Window	Logs commands entered in the Command Window.
Workspace Window	Provides information about the variables that are used.
Current Directory Window	Shows the files in the current directory.

Command Window: The Command Window is MATLAB’s main window and opens when MATLAB is started.

- Commands are typed next to the prompt (`>>`) and are executed when the **Enter** key is pressed.
- Once a command is typed and the **Enter** key is pressed, the command is executed. However, only the last command is executed. Everything executed previously is unchanged.
- Output generated by the command is displayed in the Command Window, unless a semicolon (`;`) is typed at the end.
- When the symbol `%` (percent symbol) is typed in the beginning of a line, the line is designated as a comment and is not executed.
- The `clc` command (type `clc` and press **Enter**) clears the Command Window. After working in the Command Window for a while, the display may be very long. Once the `clc` command is executed, a clear window is displayed. The command does not change anything that was done before. For example, variables that have been defined previously still exist in the memory and can be used. The up-arrow key (`↑`) can also be used to recall commands that were typed before.

Figure Window: The Figure Window opens automatically when graphics commands are executed and contains graphs created by these commands.

Editor Window: The Editor Window is used for writing and editing programs. This window is opened from the **File** menu in the Command Window. More details on the Editor Window are given in Section A.5

where it is used for creating script files.
Help Window: The Help Window contains help information. This window can be opened from the **Help** menu in the toolbar of any MATLAB window. The Help Window is interactive and can be used to obtain information on any feature of MATLAB.

Elementary arithmetic operations with scalars

The simplest way to use MATLAB is as a calculator. With scalars, the symbols of arithmetic operations are:

Operation	Symbol	Example	Operation	Symbol	Example
Addition	+	5 + 3	Right division	/	5 / 3
Subtraction	−	5 − 3	Left division	\	5 \ 3 = 3 / 5
Multiplication	*	5 * 3	Exponentiation	^	5 ^ 3 (means 5 ³ = 125)

A mathematical expression can be typed in the Command Window. When the **Enter** key is pressed, MATLAB calculates the expression and responds by displaying `ans =` and the numerical result of the expression in the next line. Examples are:

```
>> 7 + 8/2
ans =
    11
>> (7+8)/2 + 27^(1/3)
ans =
    10.5000
```

Numerical values can also be assigned to variables, which is a name made of a letter or a combination of several letters (and digits). Variable names must begin with a letter. Once a variable is assigned a numerical value, it can be used in mathematical expressions, in functions, and in any MATLAB statements and commands.

```
>> a = 12
a =
    12
>> B = 4;
>> C = (a - B) + 40 - a/B*10
C =
    18
```

← Since a semicolon is typed at the end of the command, the value of B is not displayed.

Elementary math built-in functions

In addition to basic arithmetic operations, expressions in MATLAB can include functions. MATLAB has a very large library of built-in functions. A function has a name and an argument (or arguments) in parentheses. For example, the function that calculates the square root of a

number is `sqrt(x)`. Its name is `sqrt`, and the argument is `x`. When the function is used, the argument can be a number, a variable, or a computable expression that can be made up of numbers and/or variables. Functions can also be included in arguments, as well as in expressions. The following shows examples of using the function `sqrt(x)` when MATLAB is used as a calculator with scalars.

```
>> sqrt(64)
ans =
    8
>> sqrt(50 + 14*3)
ans =
    9.5917
>> sqrt(54 + 9*sqrt(100))
ans =
    12
>> (15 + 600/4)/sqrt(121)
ans =
    15
```

Argument is a number.

Argument is an expression.

Argument includes a function.

Function is included in an expression.

Lists of some commonly used elementary MATLAB mathematical built-in functions are given in Table A-2. A complete list of functions organized by category can be found in the Help Window.

Table A-2: Built-in elementary math functions.

Command	Description	Example
<code>sqrt(x)</code>	Square root.	<pre>>> sqrt(81) ans = 9</pre>
<code>exp(x)</code>	Exponential (e^x).	<pre>>> exp(5) ans = 148.4132</pre>
<code>abs(x)</code>	Absolute value.	<pre>>> abs(-24) ans = 24</pre>
<code>log(x)</code>	Natural logarithm. Base e logarithm (ln).	<pre>>> log(1000) ans = 6.9078</pre>
<code>log10(x)</code>	Base 10 logarithm.	<pre>>> log10(1000) ans = 3.0000</pre>
<code>sin(x)</code>	Sine of angle x (x in radians).	<pre>>> sin(pi/6) ans = 0.5000</pre>
<code>sind(x)</code>	Sine of angle x (x in degrees).	<pre>>> sind(30) ans = 0.5000</pre>

Table A-2: Built-in elementary math functions. (Continued)

Command	Description	Example
The other trigonometric functions are written in the same way. The inverse trigonometric functions are written by adding the letter “a” in front, for example, <code>asin(x)</code> .		
<code>round(x)</code>	Round to the nearest integer.	<pre>>> round(17/5) ans = 3</pre>
<code>fix(x)</code>	Round toward zero.	<pre>>> fix(9/4) >> fix(-9/4) ans = ans = 2 -2</pre>
<code>ceil(x)</code>	Round up toward infinity.	<pre>>> ceil(11/5) ans = 3</pre>
<code>floor(x)</code>	Round down toward minus infinity.	<pre>>> floor(-9/4) ans = -3</pre>

Display formats

The format in which MATLAB displays output on the screen can be changed by the user. The default output format is fixed point with four decimal digits (called `short`). The format can be changed with the `format` command. Once the `format` command is entered, all the output that follows will be displayed in the specified format. Several of the available formats are listed and described in Table A-3.

Table A-3: Display format

Command	Description	Example
<code>format short</code>	Fixed point with four decimal digits for: $0.001 \leq \text{number} \leq 1000$ Otherwise display format <code>short e</code> .	<pre>>> 290/7 ans = 41.4286</pre>
<code>format long</code>	Fixed point with 14 decimal digits for: $0.001 \leq \text{number} \leq 100$ Otherwise display format <code>long e</code> .	<pre>>> 290/7 ans = 41.42857142857143</pre>
<code>format short e</code>	Scientific notation with four decimal digits.	<pre>>> 290/7 ans = 4.1429e+001</pre>
<code>format long e</code>	Scientific notation with 15 decimal digits.	<pre>>> 290/7 ans = 4.142857142857143e+001</pre>
<code>format short g</code>	Best of 5-digit fixed or floating point.	<pre>>> 290/7 ans = 41.429</pre>

Table A-3: Display format (Continued)

Command	Description	Example
<code>format long g</code>	Best of 15-digit fixed or floating point.	<pre>>> 290/7 ans = 41.4285714285714</pre>
<code>format bank</code>	Two decimal digits.	<pre>>> 290/7 ans = 41.43</pre>

A.3 ARRAYS

The array is a fundamental form that MATLAB uses to store and manipulate data. An array is a list of numbers arranged in rows and/or columns. The simplest array (one-dimensional) is a row, or a column of numbers, which in science and engineering is commonly called a vector. A more complex array (two-dimensional) is a collection of numbers arranged in rows and columns, which in science and engineering is called a matrix. Each number in a vector or a matrix is called an element. This section shows how to construct vectors and matrices. Section A.4 shows how to carry out mathematical operations with arrays.

Creating a vector

In MATLAB, a vector is created by assigning the elements of the vector to a variable. This can be done in several ways depending on the source of the information that is used for the elements of the vector. When a vector contains specific numbers that are known, the value of each element is entered directly by typing the values of the elements inside square brackets:

```
variable_name = [number number ... number]
```

For a row vector, the numbers are typed with a space or a comma between the elements. For a column vector the numbers are typed with a semicolon between them. Each element can also be a mathematical expression that can include predefined variables, numbers, and functions. Often, the elements of a row vector are a series of numbers with constant spacing. In such cases the vector can be created by typing:

```
variable_name = m:q:n
```

where *m* is the first element, *q* is the spacing, and *n* is the last element. Another option is to use the `linspace` command:

```
variable_name = linspace(xi,xf,n)
```

Several examples of constructing vectors are:

```
>> yr = [1984 1986 1988 1990 1992 1994 1996]
```

Row vector by typing elements.

```

yr =
    1984    1986    1988    1990    1992    1994    1996
>> pnt = [2; 4; 5]
pnt =
     2
     4
     5
>> x = [1:2:13]
x =
     1     3     5     7     9    11    13
>> va = linspace(0,8,6)
va =
     0    1.6000    3.2000    4.8000    6.4000    8.0000

```

Column vector by typing elements.

Row vector with constant spacing.

Row vector with 6 elements, first element 0, last element 8.

Creating a two-dimensional array (matrix)

A two-dimensional array, also called a matrix, has numbers in rows and columns. A matrix is created by assigning the elements of the matrix to a variable. This is done by typing the elements, row by row, inside square brackets []. Within each row the elements are separated with spaces or commas. Between rows type ; or the press **Enter**.

```
variable_name=[1st row elements; 2nd row elements;...; last row elements]
```

The elements that are entered can be numbers or mathematical expressions that may include numbers, predefined variables, and functions. All the rows must have the same number of elements. If an element is zero, it has to be entered as such. MATLAB displays an error message if an attempt is made to define an incomplete matrix. Examples of matrices created in different ways are:

```

>> a = [5 35 43; 4 76 81; 21 32 40]
a =
     5     35     43
     4     76     81
    21     32     40
>> cd = 6; e = 3; h = 4;
>> Mat = [e, cd*h, cos(pi/3); h^2, sqrt(h*h/cd), 14]
Mat =
     3.0000    24.0000     0.5000
    16.0000     1.6330    14.0000

```

Semicolons are typed between rows.

Variables are defined.

Elements are entered as mathematical expressions.

- All variables in MATLAB are arrays. A scalar is an array with one element; a vector is an array with one row, or one column, of elements; and a matrix is an array with elements in rows and columns.

- The variable (scalar, vector, or matrix) is defined by the input when the variable is assigned. There is no need to define the size of the array (single element for a scalar, a row or a column of elements for a vector, or a two-dimensional array of elements for a matrix) before the elements are assigned.
- Once a variable exists as a scalar, a vector, or a matrix, it can be changed to be any other size, or type, of variable. For example, a scalar can be changed to a vector or a matrix, a vector can be changed to a scalar, a vector of different length, or a matrix, and a matrix can be changed to have a different size, or to be reduced to a vector or a scalar. These changes are made by adding or deleting elements.

Array addressing

Elements in an array (either vector or matrix) can be addressed individually or in subgroups. This is useful when there is a need to redefine only some of the elements, or to use specific elements in calculations, or when a subgroup of the elements is used to define a new variable.

The address of an element in a vector is its position in the row (or column). For a vector named *ve*, *ve(k)* refers to the element in position *k*. The first position is 1. For example, if the vector *ve* has nine elements:

ve = 35 46 78 23 5 14 81 3 55

then

ve(4) = 23, *ve*(7) = 81, and *ve*(1) = 35.

The address of an element in a matrix is its position, defined by the row number and the column number where it is located. For a matrix assigned to a variable *ma*, *ma(k,p)* refers to the element in row *k* and column *p*.

For example, if the matrix is: $ma = \begin{bmatrix} 3 & 11 & 6 & 5 \\ 4 & 7 & 10 & 2 \\ 13 & 9 & 0 & 8 \end{bmatrix}$

then, *ma*(1,1) = 3, and *ma*(2,3) = 10.

It is possible to change the value of one element by reassigning a new value to the specific element. Single elements can also be used like variables in mathematical expressions.

```
>> VCT = [35 46 78 23 5 14 81 3 55]
VCT =
    35    46    78    23     5    14    81     3    55
>> VCT(4)=-2; VCT(6)=273
VCT =
    35    46    78    -2     5   273    81     3    55
>> VCT(5)^VCT(8) + sqrt(VCT(7))
```

Define a vector.

Assign new values to the fourth and sixth elements.

Use vector elements in a mathematical expression.

```

ans =
    134
>> MAT = [3 11 6 5; 4 7 10 2; 13 9 0 8]
MAT =
     3    11     6     5
     4     7    10     2
    13     9     0     8
>> MAT(3,1) = 20
MAT =
     3    11     6     5
     4     7    10     2
    20     9     0     8
>> MAT(2,4) - MAT(1,2)
ans =
    -9

```

Define a matrix.

Assign a new value to the (3,1) element.

Use matrix elements in a mathematical expression.

Using a colon : in addressing arrays

A colon can be used to address a range of elements in a vector or a matrix. If va is a vector, $va(m:n)$ refers to elements m through n of the vector va .

If A is a matrix, $A(:,n)$ refers to the elements in all the rows of column n . $A(n,:)$ refers to the elements in all the columns of row n . $A(:,m:n)$ refers to the elements in all the rows between columns m and n . $A(m:n,:)$ refers to the elements in all the columns between rows m and n . $A(m:n,p:q)$ refers to the elements in rows m through n and columns p through q .

```

>> v = [4 15 8 12 34 2 50 23 11]
v =
     4    15     8    12    34     2    50    23    11
>> u = v(3:7)
u =
     8    12    34     2    50
>> A = [1 3 5 7 9 11; 2 4 6 8 10 12; 3 6 9 12 15 18; 4 8 12 16 20
24; 5 10 15 20 25 30]
A =
     1     3     5     7     9    11
     2     4     6     8    10    12
     3     6     9    12    15    18
     4     8    12    16    20    24
     5    10    15    20    25    30
C = A(2,:)
C =
     2     4     6     8    10    12
>> F = A(1:3,2:4)

```

Define a vector.

Vector u is created from the elements 3 through 7 of vector v .

Define a matrix.

Vector C is created from the second row of matrix A .Matrix F is created from the elements in rows 1 through 3 and columns 2 through 4 of matrix A .

```
F =
     3     5     7
     4     6     8
     6     9    12
```

MATLAB has many built-in functions for managing and handling arrays. Several are listed in Table A-4.

Table A-4: Built-in functions for handling arrays.

Command	Description	Example
<code>length(A)</code>	Returns the number of elements in vector A.	<pre>>> A = [5 9 2 4]; >> length(A) ans = 4</pre>
<code>size(A)</code>	Returns a row vector $[m, n]$, where m and n are the size $m \times n$ of the array A. (m is number of rows. n is number of columns.)	<pre>>> A = [6 1 4 0 12; 5 19 6 8 2] A = 6 1 4 0 12 5 19 6 8 2 >> size(A) ans = 2 5</pre>
<code>zeros(m,n)</code>	Creates a matrix with m rows and n columns, in which all the elements are the number 0.	<pre>>> zr = zeros(3,4) zr = 0 0 0 0 0 0 0 0 0 0 0 0</pre>
<code>ones(m,n)</code>	Creates a matrix with m rows and n columns, in which all the elements are the number 1.	<pre>>> ne = ones(4,3) ne = 1 1 1 1 1 1 1 1 1 1 1 1</pre>
<code>eye(n)</code>	Creates a square matrix with n rows and n columns in which the diagonal elements are equal to 1 (identity matrix).	<pre>>> idn = eye(3) idn = 1 0 0 0 1 0 0 0 1</pre>

Strings

- A string is an array of characters. It is created by typing the characters within single quotes.
- Strings can include letters, digits, other symbols, and spaces.
- Examples of strings: 'ad ef', '3%fr2', '{edcba:21!'.

- When a string is being typed in, the color of the text on the screen changes to maroon when the first single quote is typed. When the single quote at the end of the string is typed the color of the string changes to purple.

Strings have several different uses in MATLAB. They are used in output commands to display text messages, in formatting commands of plots, and as input arguments of some functions. Strings can also be assigned to variables by simply typing the string on the right side of the assignment operator, as shown in the next example.

```
>> a = 'FRty 8'
a =
FRty 8
>> B = 'My name is John Smith'
B =
My name is John Smith
```

A.4 MATHEMATICAL OPERATIONS WITH ARRAYS

Once variables are created in MATLAB, they can be used in a wide variety of mathematical operations. Mathematical operations in MATLAB can be divided into three categories:

1. Operations with scalars ((1×1) arrays) and with single elements of arrays.
2. Operations with arrays following the rules of linear algebra.
3. Element-by-element operations with arrays.

Operations with scalars and single elements of arrays are done by using the standard symbols as in a calculator. So far, all the mathematical operations in the appendix have been done in this way.

Addition and subtraction of arrays

With arrays, the addition, subtraction, and multiplication operations follow the rules of linear algebra (see Chapter 2). The operations $+$ (addition) and $-$ (subtraction) can only be carried out with arrays of identical size (the same number of rows and columns). The sum, or the difference of two arrays, is obtained by adding, or subtracting, their corresponding elements. For example, if A and B are two (2×3) matrices,

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \end{bmatrix}$$

then, the matrix that is obtained by adding A and B is:

$$\begin{bmatrix} (A_{11} + B_{11}) & (A_{12} + B_{12}) & (A_{13} + B_{13}) \\ (A_{21} + B_{21}) & (A_{22} + B_{22}) & (A_{23} + B_{23}) \end{bmatrix}.$$

In MATLAB, when a scalar (number) is added to, or subtracted from,

an array, the number is added to, or subtracted from, all the elements of the array. Examples are:

```
>> VA = [8 5 4]; VB = [10 2 7];
>> VC = VA + VB
VC =
    18     7    11
>> A = [5 -3 8; 9 2 10], B = [10 7 4; -11 15 1]
A =
     5    -3     8
     9     2    10
B =
    10     7     4
   -11    15     1
>> C = A + B
C =
    15     4    12
    -2    17    11
>> C - 8
ans =
     7    -4     4
   -10     9     3
```

Define two vectors VA and VB.

Define a vector VC that is equal to VA + VB.

Define two matrices A and B.

Define a matrix C that is equal to A + B.

Subtract 8 from the matrix C.

8 is subtracted from each element of C.

Multiplication of arrays

The multiplication operation `*` is executed by MATLAB according to the rules of linear algebra (see Section 2.4.1). This means that if A and B are two matrices, the operation $A*B$ can be carried out only if the number of columns in matrix A is equal to the number of rows in matrix B . The result is a matrix that has the same number of rows as A and the same number of columns as B . For example, if E is a (3×2) matrix and G is a (2×4) matrix, then the operation $C=E*G$ gives a (3×4) matrix:

```
>> A = [2 -1; 8 3; 6 7], B = [4 9 1 -3; -5 2 4 6]
A =
     2    -1
     8     3
     6     7
B =
     4     9     1    -3
    -5     2     4     6
>> C = A*B
C =
    13    16    -2   -12
    17    78    20    -6
   -11    68    34    24
```

Define two matrices A and B.

Multiply A*B.

C is a (3×4) matrix.

Two vectors can be multiplied only if both have the same number of elements, and one is a row vector and the other is a column vector. The

multiplication of a row vector times a column vector gives a (1×1) matrix, which is a scalar. This is the dot product of two vectors. (MATLAB also has a built-in function, named `dot(a,b)`, that computes the dot product of two vectors.) When using the `dot` function, the vectors `a` and `b` can each be a row or a column vector. The multiplication of a column vector times a row vector, both with n elements, gives an $(n \times n)$ matrix.

```
>> AV = [2  5  1]
AV =
     2     5     1
>> BV = [3; 1; 4]
BV =
     3
     1
     4
>> AV * BV
ans =
    15
>> BV * AV
ans =
     6    15     3
     2     5     1
     8    20     4
```

Define three-element row vector AV.

Define three-element column vector BV.

Multiply AV by BV. The answer is a scalar. (Dot product of two vectors.)

Multiply BV by AV. The answer is a (3×3) matrix. (Cross product of two vectors.)

Array division

The division operation in MATLAB is associated with the solution of a system of linear equations. MATLAB has two types of array division, which are the left division and the right division. The two operations are explained in Section 4.8.1. Note that division *is not* a defined operation in linear algebra (see Section 2.4.1). The division operation in MATLAB performs the equivalent of multiplying a matrix by the inverse of another matrix (or vice versa).

Element-by-element operations

Element-by-element operations are carried out on each element of the array (or arrays). Addition and subtraction are already by definition element-by-element operations because when two arrays are added (or subtracted) the operation is executed with the elements that are in the same position in the arrays. In the same way, multiplication, division, and exponentiation can be carried out on each element of the array. When two or more arrays are involved in the same expression, 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 are entered in MATLAB by typing a period in

front of the arithmetic operator.

Symbol	Description	Symbol	Description
.*	Multiplication	./	Right division
.^	Exponentiation	.\	Left Division

If two vectors a and b are $a = [a_1 \ a_2 \ a_3 \ a_4]$ and $b = [b_1 \ b_2 \ b_3 \ b_4]$, then element-by-element multiplication, division, and exponentiation of the two vectors are:

$$\begin{aligned} a .* b &= [a_1 b_1 \ a_2 b_2 \ a_3 b_3 \ a_4 b_4] \\ a ./ b &= [a_1 / b_1 \ a_2 / b_2 \ a_3 / b_3 \ a_4 / b_4] \\ a .^ b &= [(a_1)^{b_1} \ (a_2)^{b_2} \ (a_3)^{b_3} \ (a_4)^{b_4}] \end{aligned}$$

If two matrices A and B are:

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix}$$

then element-by-element multiplication and division of the two matrices give:

$$A .* B = \begin{bmatrix} A_{11} B_{11} & A_{12} B_{12} & A_{13} B_{13} \\ A_{21} B_{21} & A_{22} B_{22} & A_{23} B_{23} \\ A_{31} B_{31} & A_{32} B_{32} & A_{33} B_{33} \end{bmatrix} \quad A ./ B = \begin{bmatrix} A_{11} / B_{11} & A_{12} / B_{12} & A_{13} / B_{13} \\ A_{21} / B_{21} & A_{22} / B_{22} & A_{23} / B_{23} \\ A_{31} / B_{31} & A_{32} / B_{32} & A_{33} / B_{33} \end{bmatrix}$$

Examples of element-by-element operations with MATLAB are:

```
>> A = [2 6 3; 5 8 4]
```

Define a (2 × 3) matrix A.

```
A =  
    2     6     3  
    5     8     4
```

```
>> B = [1 4 10; 3 2 7]
```

Define a (2 × 3) matrix B.

```
B =  
    1     4    10  
    3     2     7
```

```
>> A .* B
```

Element-by-element multiplication of arrays A and B.

```
ans =  
    2    24    30  
   15    16    28
```

```
>> C = A ./ B
```

Element-by-element division of array A by array B.

```
C =
    2.0000    1.5000    0.3000
    1.6667    4.0000    0.5714
>> B .^ 3
ans =
     1    64   1000
    27     8    343
```

Element-by-element exponentiation of array B.

Element-by-element calculations are very useful for calculating the value of a function at many values of its argument. This is done by first defining a vector that contains values of the independent variable and then by using this vector in element-by-element computations to create a vector in which each element is the corresponding value of the function. For example, calculating $y = \frac{z^3 + 5z}{4z^2 - 10}$ for eight values of z , $z = 1, 3, 5, \dots, 15$, is accomplished as follows:

```
>> z = [1:2:15]
z =
     1     3     5     7     9    11    13    15
>> y = (z.^3 + 5*z) ./ (4*z.^2 - 10)
y =
-1.0000    1.6154    1.6667    2.0323    2.4650    2.9241    3.3964    3.8764
```

Define a vector z with eight elements.

Vector z is used in element-by-element calculation of the elements of vector y.

In the last example element-by-element operations are used three times; to calculate z^3 and z^2 and to divide the numerator by the denominator. MATLAB has many built-in functions for operations with arrays. Several of these functions are listed in Table A-5.

Table A-5: Built-in functions for handling arrays.

Command	Description	Example
mean (A)	If A is a vector, the function returns the mean value of the elements of the vector.	>> A = [5 9 2 4] ; >> mean (A) ans = 5
sum (A)	If A is a vector, the function returns the sum of the elements of the vector.	>> A = [5 9 2 4] ; >> sum (A) ans = 20
sort (A)	If A is a vector, the function arranges the elements of the vector in ascending order.	>> A = [5 9 2 4] ; >> sort (A) ans = 2 4 5 9

Table A-5: Built-in functions for handling arrays. (Continued)

Command	Description	Example
det (A)	The function returns the determinant of a square matrix A.	<pre>>> A = [2 4; 3 5]; >> det (A) ans = -2</pre>

A.5 SCRIPT FILES

A script file is a file that contains a sequence of MATLAB commands, which is also called a program. When a script file is run, MATLAB executes the commands in the order they are written just as if they were typed in the Command Window. When a command generates output (e.g., assignment of a value to a variable without semicolon at the end), the output is displayed in the Command Window. Using a script file is convenient because it can be stored, edited later (corrected and/or changed), and executed many times. Script files can be typed and edited in any text editor and then pasted into the MATLAB editor. Script files are also called M-files because the extension `.m` is used when they are saved.

Creating and saving a script file

Script files are created and edited in the Editor/Debugger Window. This window is opened from the Command Window. In the **File** menu, select **New** and then select **M-file**. Once the window is open, the commands of the script file are typed line by line. MATLAB automatically numbers a new line every time the **Enter** key is pressed. The commands can also be typed in any text editor or word processor program and then copied and pasted in the Editor/Debugger Window.

Before a script file can be executed, it has to be saved. This is done by choosing **Save As...** from the **File** menu, selecting a location (folder), and entering a name for the file. The rules for naming a script file follow the rules of naming a variable (must begin with a letter, can include digits and underscore, and be up to 63 characters long). The names of user-defined variables, predefined variables, MATLAB commands, or functions should not be used to name script files.

A script file can be executed either by typing its name in the Command Window and then pressing the **Enter** key, or directly from the Editor Window by clicking on the **Run** icon. Before this can be done, however, the user has to make sure that MATLAB can find the file (i.e., that MATLAB knows where the file is saved). In order to be able to run a file, the file must be in either the current directory or the search path.

The current directory is shown in the “Current Directory” field in the desktop toolbar of the Command Window. The current directory can be changed in the Current Directory Window.

When MATLAB is asked to run a script file or to execute a func-

tion, it searches for the file in directories listed in the search path. The directories included in the search path are displayed in the Set Path Window that can be opened by selecting *Set Path* in the *File* menu. Once the Set Path Window is open, new folders can be added to, or removed from, the search path.

Input to a script file

When a script file is executed, the variables used in the calculations within the file must have assigned values. The assignment of a value to a variable can be done in three ways, depending on where and how the variable is defined. One option is to define the variable and assign it a value in the script file. In this case the assignment of value to the variable is part of the script file. If the user wants to run the file with a different variable value, the file must be edited and the assignment of the variable changed. Then, after the file is saved, it can be executed again.

A second option is to define the variable and assign it a value in the Command Window. In this case, if the user wants to run the script file with a different value for the variable, the new value is assigned in the Command Window and the file is executed again.

The third option is to define the variable in the script file but assign a specific value in the Command Window when the script file is executed. This is done by using the `input` command.

Output from a script file

As discussed earlier, MATLAB automatically generates a display when some commands are executed. For example, when a variable is assigned a value, or the name of a previously assigned variable is typed and the **Enter** key is pressed, MATLAB displays the variable and its value. In addition, MATLAB has several commands that can be used to generate displays. The displays can be messages that provide information, numerical data, and plots. Two commands frequently used to generate output are `disp` and `fprintf`. The `disp` command displays the output on the screen, while the `fprintf` command can be used to display the output on the screen or to save the output to a file.

The `disp` command is used to display the elements of a variable without displaying the name of the variable and to display text. The format of the `disp` command is:

`disp(name of a variable) or disp('text as string')`

Every time the `disp` command is executed, the display it generates appears in a new line.

The `fprintf` command can be used to display output (text and data) on the screen or to save it to a file. With this command the output can be formatted. For example, text and numerical values of variables can be intermixed and displayed in the same line. In addition, the format of the numbers can be controlled. To display a mix of text and a

number (value of a variable), the `fprintf` command has the form:

```
fprintf('text as string %-5.2f additional text', variable_name)
```

The % sign marks the spot where the number is inserted within the text.

Formatting elements (define the format of the number).

The name of the variable whose value is displayed.

A.6 PLOTTING

MATLAB has many commands that can be used for creating different types of plots. These include standard plots with linear axes, plots with logarithmic axes, bar and stairs plots, polar plots, and many more. The plots can be formatted to have a desired appearance.

Two-dimensional plots can be created with the `plot` command. The simplest form of the command is:

```
plot(x,y)
```

The arguments `x` and `y` are each a vector (one-dimensional array). Both vectors must have the same number of elements. When the `plot` command is executed, a figure appears in the Figure Window, which opens automatically. The figure has a single curve with the `x` values on the abscissa (horizontal axis) and the `y` values on the ordinate (vertical axis). The curve is constructed of straight line segments that connect the points whose coordinates are defined by the elements of the vectors `x` and `y`. The vectors, of course, can have any name. The vector that is typed first in the `plot` command is used for the horizontal axis, and the vector that is typed second is used for the vertical axis. The figure that is displayed has axes with linear scale and default range. For example, if a vector `x` has the elements 1, 2, 3, 5, 7, 7.5, 8, 10, and a vector `y` has the elements 2, 6.5, 7, 7, 5.5, 4, 6, 8, a simple plot of `y` versus `x` can be produced by typing the following in the Command Window:

```
>> x = [1 2 3 5 7 7.5 8 10];  
>> y = [2 6.5 7 7 5.5 4 6 8];  
>> plot(x,y)
```

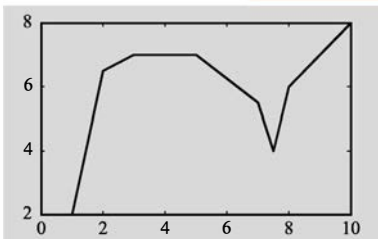


Figure A-1: A plot of data points.

Once the `plot` command is executed, the plot that is shown in Fig. A-5 is displayed in the Figure Window.

The `plot` command has additional optional arguments that can be used to specify the color and style of the line and the color and type of markers, if any are desired. With these options the command has the

form:

plot(x,y,'line specifiers')

Line specifiers can be used to define the style and color of the line and the type of markers (if markers are desired). The line style specifiers are:

Line Style	Specifier	Line Style	Specifier
solid (default)	-	dotted	:
dashed	--	dash-dot	-.

The line color specifiers are:

Line Color	Specifier	Line Style	Specifier	Line Color	Specifier	Line Color	Specifier
red	r	blue	b	magenta	m	black	k
green	g	cyan	c	yellow	y	white	w

The marker type specifiers are:

Marker	Specifier	Marker	Specifier	Marker	Specifier
plus sign	+	asterisk	*	square	s
circle	o	point	.	diamond	d

The specifiers are typed inside the `plot` command as strings. Within the string the specifiers can be typed in any order.

The `plot` command creates bare plots. The plot can be modified to include axis labels, a title, and other features. Plots can be formatted by using MATLAB commands that follow the `plot` command. The formatting commands for adding axis labels and a title are:

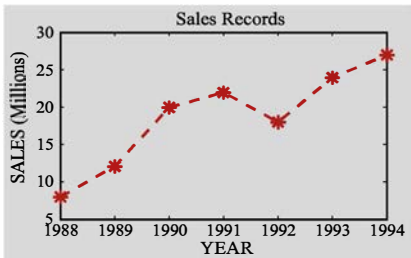


Figure A-2: Formatted plot.

xlabel('text as string')
ylabel('text as string')
title('text as string')

For example, the program listed below produces the plot that is displayed in Fig. A-2. The data is plotted with a dashed red line and asterisk markers, and the figure includes axis labels and a title.

```
>>yr=[1988:1:1994];  
>>sle=[8 12 20 22 18 24 27];  
>>plot(yr,sle,'--r*','linewidth',2,'markersize',12)  
>>xlabel('YEAR')  
>>ylabel('SALES (Millions)')  
>>title('Sales Records')
```

Formatting of plots can also be done in the Figure Window using the insert menu or Plot Editor.

A.7 USER-DEFINED FUNCTIONS AND FUNCTION FILES

A function in mathematics can be expressed in the form $y = f(x)$, where $f(x)$ is a mathematical expression in terms of x . A value of y (output) is obtained when a value of x (input) is substituted in the expression. A function file in MATLAB is a computer program that is used like a math function. Data is imported into the program and is used for calculating the value of the function. Schematically, a function file can be illustrated by:



The input and output arguments can be one or several variables, and each can be a scalar, a vector, or an array of any size. Functions can be used for a math function and as subprograms in large programs. In this way large computer programs can be made up of smaller “building blocks” that can be tested independently.

MATLAB has already many built-in functions. Examples are the standard math functions (i.e. `sin(x)`, `cos(x)`, `sqrt(x)`, and `exp(x)`) that are used by typing their name with a value for the input argument. MATLAB contains also many built-in functions for executing more complicated operations (e.g., solving a nonlinear equation, curve fitting).

MATLAB users can write additional (new) user-defined functions that can be used like the built-in functions. This section describes how to write, save, and use user-defined functions.

User-defined functions are created and edited, like script files, in the Editor/Debugger Window. The first executable line in a function file must be the function definition line that has the form:

```
function [output arguments] = function_name(input arguments)
```

The word `function` must be the first word and must be typed in lower case letters.

A list of output arguments typed inside brackets and separated by commas.

The name of the function.

A list of input arguments typed inside parentheses and separated by commas.

The word “`function`”, typed in lower case letters, must be the first word in the function definition line. The input and output arguments are

used to transfer data into and out of the function. The input arguments are listed inside parentheses following the function name. Usually, there is at least one input argument. If there are more than one, the input arguments are separated by commas. The computer code that performs the calculations within the function file is written in terms of the input arguments and assumes that the arguments have assigned numerical values.

The output arguments, which are listed inside brackets on the left side of the assignment operator in the function definition line, transfer the output from the function file. A user-defined function can have one, several, or no output arguments. If there are more than one, the output arguments are separated with commas or spaces. If there is only one output argument, it can be typed without brackets. In order for the user-defined function to work, the output arguments must be assigned values within the computer program of the function.

Following the function definition line, there are usually several lines of comments. They are optional but frequently used to provide information about the function. Next, the function contains the computer program (code) that actually performs the computations. The code can use all MATLAB programming features, including calculations, assignments, any built-in or user-defined functions, and flow control (conditional statements and loops; see Section A.11).

All the variables in a user-defined function are local. This means that the variables that are defined within the program of the user-defined function are recognized only in this program. When a function file is executed, MATLAB uses an area of memory that is separate from the workspace (the memory space of the Command Window and the script files). In a user-defined function the input variables are assigned values each time the function is called. These variables are then used in the calculations within the function file. When the function file finishes its execution, the values of the output arguments are transferred to the variables that were used when the function was called. Thus, a function file can have variables with the same name as variables in the Command Window or in script files. The function file does not recognize variables with the same name that have been assigned values outside the function. The assignment of values to these variables in the function file will not change their assignment elsewhere.

A simple user-defined function, named `loan`, that calculates the monthly and total pay of a loan for a given loan amount, interest rate, and duration is listed next.

```
function [mpay,tpay] = loan(amount,rate,years) Function definition line.
%loan calculates monthly and total payment of loan.
%Input arguments:
%amount:  loan amount in $.
%rate:    annual interest rate in percent.
```

```

%years:  number of years.
%Output arguments:
%mpay:  monthly payment.
%tpay:  total payment.

format bank
ratem=rate*0.01/12;
a=1+ratem;
b=(a^(years*12)-1)/ratem;
mpay=amount*a^(years*12)/(a*b);
tpay=mpay*years*12;

```

Assign values to the output arguments.

The user-defined function `loan` is next used in the Command Window for calculating the monthly and total pay of a four-year loan of \$25,000 with interest rate of 4%:

```

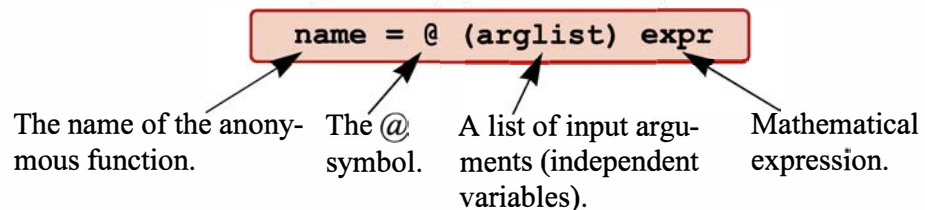
>> [month total] = loan(25000,7.5,4)
month =
    600.72
total =
   28834.47

```

A.8 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 defined in the Command Window, within a script file, or inside a user-defined function.

An anonymous function is created by typing the following command:



A simple example is: `cube = @ (x) x^3`, which calculates the cubic power of the input argument.

- The command creates the anonymous function, and assigns a handle for the function to the variable name that is on the left-hand side of the `=` sign. Function handles provide means for referencing the function, and passing it to other functions, see Section A.8.

- The `expr` consists of a single valid mathematical MATLAB expression.
- The mathematical expression can have one or several independent variables. The independent variable(s) is (are) listed in the `(arglist)`. If there are more than one, the independent variables are separated with commas.
An example of an anonymous function that has two independent variables is: `circle = @(x,y) 16*x^2+9*y^2`
- The mathematical expression can include any built-in or user-defined functions.
- The operations in the mathematical expression must be written according to the dimensions of the arguments (element-by-element or linear algebra calculations).
- The expression can include predefined variables that are already defined when the anonymous function is defined. For example, if three variables `a`, `b`, and `c` are defined (they have assigned numerical values), then they can be used in the expression of the anonymous function: `parabola = @(x) a*x^2+b*x+c`. **Important note:** MATLAB captures the values of the predefined variables when the anonymous function is defined. This means that if subsequently new values are assigned to the predefined variables, the anonymous function is not changed. The anonymous function has to be redefined in order for the new values of the predefined variables to be used in the expression of the anonymous function.

Using an anonymous function

- 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).
- Anonymous functions can also be used as arguments in other functions (see Section A.8).

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 displays the function. The function can then be used for different values of `x`:

```
>> FA(2)
```

```
ans =
    18.1994
>> z = FA(3)
z =
    2.1656e+003
```

If x is expected to be an array, and 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
```

Using a vector as input argument.

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
```

A.9 FUNCTION FUNCTIONS

There are many situations where a function has to be imported into another function. For example, MATLAB has a built-in function called `fzero` that finds the zero of a math function $f(x)$, i.e., the value of x where $f(x) = 0$. The program (code) of the function `fzero` is written in such a way that it can find the zero of different functions. When `fzero` is used, the specific function to be solved is passed (imported) into `fzero`. (The function `fzero` is described in detail in Chapter 3.)

A function function is a MATLAB function (built-in, or user-defined) that imports another function as an input argument. A function function includes in its input arguments a name (a dummy function name) that represents the imported function. The dummy function name is used in the operations within the program (code) of the function function. When the function function is used (called), the specific function

that is imported is listed in its input argument by using a function handle.

Function handle

A function handle is a MATLAB value that is associated to a function. It is a MATLAB data type, and can be passed as an argument into another function. Once passed, the function handle provides means for calling (using) the function it is associated with. Function handles can be used with any kind of MATLAB function. This includes built-in functions, user-defined functions, and anonymous functions.

- For built-in and user-defined functions, a function handle is created by typing the symbol `@` in front of the function name. For example, `@cos` is the function handle of the built-in function `cos`, and `@loan` is the function handle of the user-defined function `loan` that was written in Section A.7.
- The function handle can also be assigned to a variable name. For example, `cosHandle=@cos` assigns the handle `@cos` to `cosHandle`. Then, the name `cosHandle` can be used for passing the handle.
- For anonymous functions (Section A.8), their name is already a function handle.

Writing a function function that accepts a function handle as an input argument

As already mentioned, the input arguments of a function function (a function that accepts another function) includes a name (dummy function name) that represents the imported function. This dummy function name (including a list of input arguments enclosed in parentheses) is used for the operations in the program within the function function.

- The function that is actually being imported must be consistent with the way that the dummy function is being used in the program. This means that both must have the same number and type of input and output arguments.

The following is an example of a user-defined function function, named `funplot`, that makes a plot of a function (any function $f(x)$ that is imported into it) between the points $x = a$ and $x = b$. The input arguments are `(Fun, a, b)`, where `Fun` is a dummy name that represents the imported function, and `a` and `b` are the endpoints of the plot. The function `funplot` also has a numerical output `xyout`, which is a 3×2 matrix with the values of x and $f(x)$ at the three points: $x = a$, $x = (a + b)/2$ and $x = b$. Note that in the program, the dummy function `Fun` has one input argument (x) and one output argument y , which are both vectors.

A name for the function that is imported (dummy function name).

```
function xyout=funplot(Fun,a,b)
% funplot makes a plot of the function Fun which is passed in
% when funplot is called in the domain [a, b].
% Input arguments:
% Fun: Function handle of the function to be plotted.
% a: The first point of the domain.
% b: The last point of the domain.
% Output argument:
% xyout: The values of x and y at x=a, x=(a+b)/2, and x=b
% listed in a 3 by 2 matrix.

x=linspace(a,b,100);
y=Fun(x);
xyout(1,1)=a; xyout(2,1)=(a+b)/2; xyout(3,1)=b;
xyout(1,2)=y(1);
xyout(2,2)=Fun((a+b)/2);
xyout(3,2)=y(100);
plot(x,y)
xlabel('x'), ylabel('y')
```

Using the imported function to calculate $f(x)$ at 100 points.

Using the imported function to calculate $f(x)$ at the midpoint.

As an example, the function `funplot` is used for making a plot of the math function $f(x) = e^{-0.17x^3} - 2x^2 + 0.8x - 3$ over the domain $[0.5, 4]$. This is demonstrated in two ways: first, by writing a user-defined function for $f(x)$, and then by writing $f(x)$ as an anonymous function.

Passing a user-defined function into a function function:

First, a user-defined function named `Fdemo` is written for $f(x)$. `Fdemo` calculates $f(x)$ for a given value of x and is written using element-by-element operations.

```
function y=Fdemo(x)
y=exp(-0.17*x).*x.^3-2*x.^2+0.8*x-3;
```

Next, the function `Fdemo` is passed into the user-defined function `funplot` which is called in the Command Window. Note that a handle of the user-defined function `Fdemo` is entered (the handle is `@Fdemo`) for the input argument `Fun` in the user-defined function `funplot`.

```
>> ydemo=funplot(@Fdemo,0.5,4)
ydemo =
    0.5000    -2.9852
    2.2500    -3.5548
    4.0000     0.6235
```

Enter a handle of the user-defined function `Fdemo`.

When the command is executed the numerical output is displayed in the Command Window, and the plot shown in the fig. A-3 is displayed in the Figure Window.

Passing an anonymous function into a function function:

To use an anonymous function, the function $f(x) = e^{-0.17x}x^3 - 2x^2 + 0.8x - 3$ first has to be written as an anonymous function, and then passed into the user-defined function `funplot`. The following shows how both of these steps are done in the Command Window. Note that the name of the anonymous function `FdemoAnony` is entered without the sign `@` for the input argument `Fun` in the user-defined function `funplot` (since the name is already the handle of the anonymous function).

Figure A-3: Plot created by `funplot`.

```
>> FdemoAnony=@(x) exp(-0.17*x).*x.^3-2*x.^2+0.8*x-3
FdemoAnony =
    @(x) exp(-0.17*x).*x.^3-2*x.^2+0.8*x-3

>> ydemo=funplot(FdemoAnony,0.5,4)
ydemo =
    0.5000    -2.9852
    2.2500    -3.5548
    4.0000     0.6235
```

Create an anonymous function for $f(x)$.

Enter the name of the anonymous function (`FdemoAnony`).

A.10 SUBFUNCTIONS

A function file can contain more than one user-defined function. The functions are typed one after the other. Each function begins with a function definition line. The first function is called the primary function and the rest of the functions are called subfunctions. The subfunctions can be typed in any order. The name of the function file that is saved should correspond to the name of the primary function. Each of the functions in the file can call any of the other functions in the file. Outside functions, or programs (script files), can only call the primary function. Each of the functions in the file has its own workspace, which means that in each the variables are local. In other words, the primary function and the subfunctions cannot access each others variables (unless variables are declared to be global).

Subfunctions can help writing user-defined functions in an organized manner. The program in the primary function can be divided into smaller tasks, where each is carried out in a subfunction. This is demonstrated in the following user-defined function named `SortAveSD`. The input to the function is a vector with numbers (grades). The function sorts the vector from the smallest element to the largest, and calculates the average and the standard deviation of the grades. The function contains three subfunctions. The average x_{ave} (mean) of a given set of n numbers x_1, x_2, \dots, x_n is given by:

$$x_{ave} = (x_1 + x_2 + \dots + x_n)/n$$

The standard deviation is given by:

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - x_{ave})^2}{n-1}}$$

The primary function.

```
function [GrSort GrAve GrSD] = SortAveSD(Grades)
% SortAveSD sorts a vector that contains grades from the small-
% est to the
% largest, calculates the average grade ans the standard devia-
% tion.
% Input argument:
% Grades A vector with the grades.
% Output arguments:
% GrSort A vector with the grades sorted from the smallest to
% the largest.
% GrAve The average grade.
% GrSD The standard deviation.
n= length(Grades);
GrAve=Ave(Grades,n);
GrSort=LowToHigh(Grades,n);
GrSD=StandDiv(Grades,n,GrAve);
```

The subfunctions are used in the primary function.

```
function av = Ave(x,num)
av=sum(x)/num;
```

A subfunction that calculates the average (mean) of the elements of a vector.

```
function xsorted = LowToHigh(x,num)
for i = 1:num-1
    for j = i+1:num
        if x(j) < x(i)
            temp=x(i);
            x(i)= x(j);
            x(j)= temp;
        end
    end
end
xsorted = x;
```

A subfunction that sorts the elements of a vector from the smallest to the largest.

```
function Sdiv = StandDiv(x,num,ave)
xdif2 = (x-ave).^2;
Sdiv = sqrt(sum(xdif2)/(num-1));
```

A subfunction that calculates the standard deviation of the elements of a vector.

Next, The user-defind function SortAveSD is used (called) in the

Command Window:

```
>> ClassGrades = [80 75 91 60 79 89 65 80 95 50 81];
>> [G A S] = SortAveSD(ClassGrades)
G =
    50    60    65    75    79    80    80    81    89    91    95
A =
    76.8182
S =
    13.6661
```

Define a vector with a list of grades.

The grades are sorted.

The average grade.

The standard deviation.

A.11 PROGRAMMING IN MATLAB

A computer program is a sequence of computer commands. In a simple program the commands are executed one after the other in the order that they are typed. Many situations, however, require more sophisticated programs in which different commands (or groups of commands) are executed when the program is executed with different input variables. In other situations there might be a need to repeat a sequence of commands several times within a program. For example, programs that solve equations numerically repeat a sequence of calculations until the error in the answer is smaller than some measure.

MATLAB provides several tools that can be used to control the flow of a program. Conditional statements make it possible to skip commands or to execute specific groups of commands in different situations. For loops and while loops make it possible to repeat a sequence of commands several times.

Changing the flow of a program requires some kind of decision-making process within the program. The computer must decide whether to execute the next command or to skip one or more commands and continue at a different line in the program. The program makes these decisions by comparing values of variables. This is done by using relational and logical operators.

A.11.1 Relational and Logical Operators

Relational and logical operators are used in combination with other commands in order to make decisions that control the flow of a computer program. A relational operator compares two numbers by determining whether a comparison statement (e.g., $5 < 8$) is true or false. If the statement is true, it is assigned a value of 1. If the statement is false, it is assigned a value of 0. Relational operators in MATLAB are given in the table that follows.

Relational Operator	Description	Relational Operator	Description
<	Less than	>=	Greater than or equal to

Relational Operator	Description	Relational Operator	Description
>	Greater than	==	Equal to
<=	Less than or equal to	~=	Not equal to

Note that the “equal to” relational operator consists of two = signs (with no space between them), since one = sign is the assignment operator. In all relational operators that consist of two characters there is no space between the characters (<=, >=, ~=). Two examples are:

```
>> 5 > 8
ans =
    0
>> 4 == 6
ans =
    0
```

A logical operator examines true/false statements and produces a result that is true (1) or false (0) according to the specific operator. Logical operators in MATLAB are:

Logical operator	Name	Description
& Example: A&B	AND	Operates on two operands (A and B). If both are true, the result is true (1); otherwise the result is false (0).
 Example: A B	OR	Operates on two operands (A and B). If either one, or both are true, the result is true (1); otherwise (both are false) the result is false (0).
~ Example: ~A	NOT	Operates on one operand (A). Gives the opposite of the operand. True (1) if the operand is false, and false (0) if the operand is true.

Logical operators can have numbers as operands. A nonzero number is true, and a zero is false. Several examples are:

```
>> 3 & 7
ans =
    1
>> a = 5 | 0
a =
    1
>> ~25
ans =
    0
```

3 and 7 are both true (nonzero), so the outcome is 1.

1 is assigned to a since at least one number is true (nonzero).

A.11.2 Conditional Statements, if-else Structures

A conditional statement is a command that allows MATLAB to make a decision of whether to execute a group of commands that follow the

conditional statement, or to skip these commands. In a conditional statement, a conditional expression is stated. If the expression is true, a group of commands that follow the statement is executed. If the expression is false, the computer skips the group.

The if-end structure

The simplest form of a conditional statement is the `if-end` structure, which is shown schematically in Fig. A-4. The figure shows how the commands are typed in the program and presents a flowchart that symbolically shows the flow, or the sequence, in which the commands are executed. As the program executes, it reaches the `if` statement. If the conditional expression in the `if` statement is true (1), the program continues to execute the commands that follow the `if` statement all the

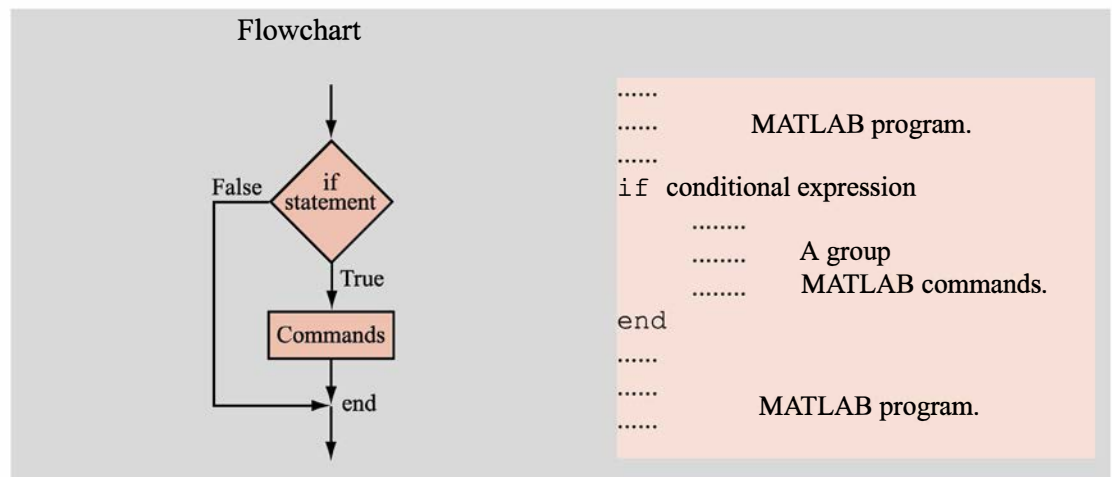


Figure A-4: The structure of the `if-end` conditional statement.

way down to the `end` statement. If the conditional expression is false (0), the program skips the group of commands between the `if` and the `end`, and continues with the commands that follow the `end` statement.

The if-else-end structure

The `if-else-end` structure provides a means for choosing one group of commands, out of a possible two groups, for execution (see Fig. A-5). The first line is an `if` statement with a conditional expression. If the conditional expression is true, the program executes the group 1 commands between the `if` and the `else` statements, and then skips to the `end`. If the conditional expression is false, the program skips to the `else` statement and executes the group 2 commands between the `else` and the `end` statements.

The if-elseif-else-end structure

The `if-elseif-else-end` structure is shown in Fig. A-6. This structure includes two conditional statements (`if` and `elseif`) that

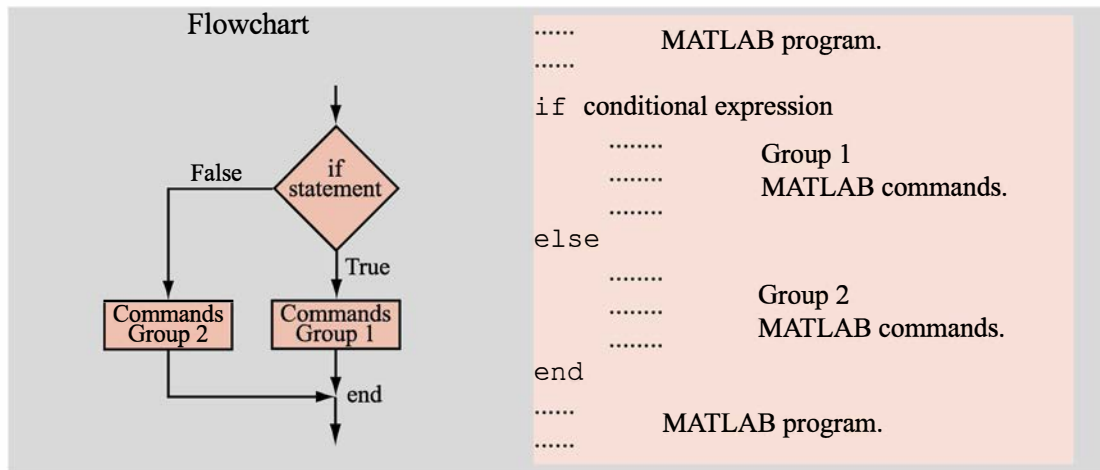


Figure A-5: The structure of the `if-else-end` conditional statement.

make it possible to select one out of three groups of commands for execution. The first line is an `if` statement with a conditional expression. If the conditional expression is true, the program executes the group 1 commands between the `if` and the `elseif` statements and then skips to the `end`. If the conditional expression in the `if` statement is false, the program skips to the `elseif` statement. If the conditional expression in the `elseif` statement is true, the program executes the group 2 commands between the `elseif` and the `else` statements and then skips to the `end`. If the conditional expression in the `elseif` statement is false, the program skips to the `else` statement and executes the group 3 commands between the `else` and the `end` statements.

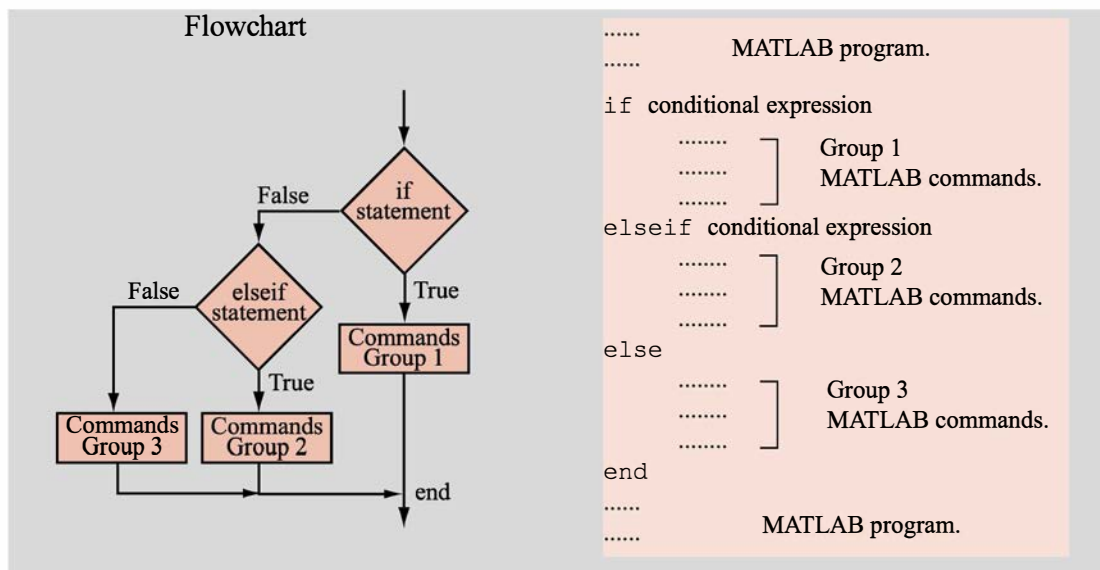


Figure A-6: The structure of the `if-elseif-else-end` conditional statement.

Several `elseif` statements and associated groups of commands can be added. In this way more conditions can be included. Also, the `else` statement is optional. This means that in the case of several `elseif` statements and no `else` statement, if any of the conditional statements is true, the associated commands are executed, but otherwise nothing is executed.

In general, the same task can be accomplished by using several `elseif` statements or `if-else-end` structures. A better programming practice is to use the latter method, which makes the program easier to understand, modify, and debug.

A.11.3 Loops

A loop is another means to alter the flow of a computer program. In a loop, the execution of a command, or a group of commands, is repeated several times consecutively. Each round of execution is called a pass. In each pass at least one variable (but usually more than one) that is defined within the loop is assigned a new value.

for-end loops

In `for-end` loops, the execution of a command or a group of commands is repeated a predetermined number of times. The form of the loop is shown in Fig. A-7. In the first pass $k = f$, and the computer exe-

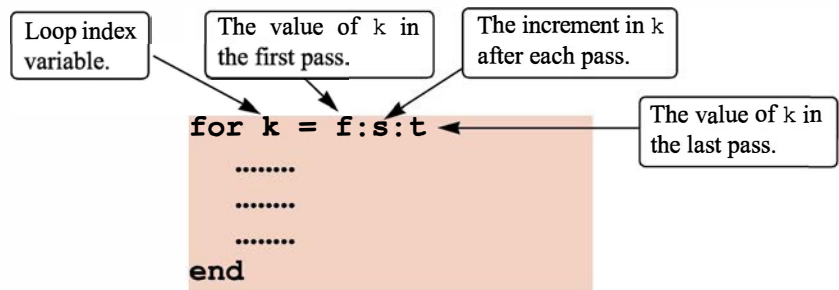


Figure A-7: The structure of a `for-end` loop.

cutes the commands between the `for` and the `end` commands. Then, the program goes back to the `for` command for the second pass. k obtains a new value equal to $k = f + s$, and the commands between the `for` and the `end` commands are executed with the new value of k . The process repeats itself until the last pass where $k = t$. Then the program does not go back to the `for`, but continues with the commands that follow the `end` command. For example, if $k = 1:2:9$, then there are five loops, and the value of k in the passes is 1, 3, 5, 7, and 9. If the increment value s is omitted, its value is 1 (default) (i.e., $k = 3:7$ produces five passes with $k = 3, 4, 5, 6, 7$).

A program that illustrates the use of conditional statements and loops is shown next (script file). The program changes the elements of a

given vector such that elements that are positive and are divisible by 3 and/or by 5 are doubled. Elements that are negative but greater than -5 are raised to the power of 3, and all the other elements are unchanged.

```
V = [5, 17, -3, 8, 0, -7, 12, 15, 20 -6, 6, 4, -2, 16];
n = length(V);
for k = 1:n
    if V(k) > 0 & (rem(V(k), 3) == 0 | rem(V(k), 5) == 0)
        V(k) = 2*V(k);
    elseif V(k) < 0 & V(k) > -5
        V(k) = V(k)^3;
    end
end
V
```

In the k th pass of the loop the k th element is checked and changed, if needed.

When the program is executed, the following new vector V is displayed in the Command Window:

```
V =
    10    17   -27     8     0    -7    24    30    40    -6    12     4    -8    16
```

A.12 PROBLEMS

A.1 Define the variables x and z as $x = 5.3$, and $z = 7.8$, then evaluate:

$$(a) \frac{xz}{(x/z)^2} + 14x^2 - 0.8z^2 \quad (b) \quad x^2z - z^2x + \left(\frac{x}{z}\right)^2 - \left(\frac{z}{x}\right)^{1/2}$$

A.2 Define two variables: $\alpha = 35^\circ$, $\beta = 23^\circ$. Using these variables, show that the following trigonometric identity is correct by calculating the value of the left and right sides of the equation:

$$\cos \alpha \cos \beta = \frac{1}{2} [\cos(\alpha - \beta) + \cos(\alpha + \beta)]$$

A.3 Two trigonometric identities are given by:

$$(a) \quad \tan 4x = \frac{4 \tan x - 4 \tan^3 x}{1 - 6 \tan^2 x + \tan^4 x} \quad (b) \quad \tan \frac{x}{2} = \frac{1 - \cos x}{\sin x}$$

For each part, verify that the identity is correct by calculating the values of the left and right sides of the equation, substituting $x = 17^\circ$.

A.4 Create a row vector with 15 equally spaced elements in which the first element is 9 and the last element is 44.

A.5 Create a column vector in which the first element is 14, the elements decrease with increments of -3, and the last element is -10. (A column vector can be created by the transpose of a row vector.)

A.6 Given: $\int \sin^2 x dx = \frac{1}{2}x - \frac{1}{4}\sin 2x$. Use MATLAB to calculate the following definite integral:

$$\int_{\frac{\pi}{3}}^{\frac{3\pi}{4}} \sin^2 x dx$$

A.7 Create the matrix shown by using the vector notation for creating vectors with constant spacing when entering the rows (i.e., do not type individual elements).

```
A =
    2.5000    3.5000    4.5000    5.5000    6.5000    7.5000
   42.0000   38.6000   35.2000   31.8000   28.4000   25.0000
   15.0000   14.6000   14.2000   13.8000   13.4000   13.0000
    3.0000    2.0000    1.0000         0   -1.0000   -2.0000
```

A.8 Create the matrix A in Problem A.7, and then use colons to address a range of elements to create the following vectors:

- Create a four-element row vector named v_a that contains the third through sixth elements of the second row of A .
- Create a three-element column vector named v_b that contains the second through fourth elements of the fifth column of A .

A.9 Create the matrix A in Problem A.7, and then use colons to address a range of elements to create the following matrices:

- Create a 3×4 matrix B from the first, second, and fourth rows, and the first, second, fourth, and sixth columns of the matrix A .
- Create a 2×3 matrix C from the second and fourth rows, and the second, fifth, and sixth columns of the matrix A .

A.10 For the function $y = \frac{(2x^2 - 16x + 4)^2}{x + 15}$, calculate the value of y for the following values of x : $-1.2, -0.4, 0.4, 1.2, 2, 2.8, 3.6$. Solve the problem by first creating a vector x , and then creating a vector y , using element-by-element calculations. Make a plot of the points using asterisk markers for the points and a black line connecting the points. Label the axes.

A.11 Define a and b as scalars $a = 3$ and $b = -4$, and x as the vector $x = -3, -2.8, -2.6, \dots, 1.6, 1.8, 2$. Then use these variables to calculate y by: $y = 8 \frac{a^2/b^3}{x^2 + b^2/a^3}$. Plot y versus x .

A.12 For the function $y = 6t^{(1/3)} - \frac{(t+3)^2}{2(t+4)} + 2$, calculate the value of y for the following values of t : $0, 2, 4, 6, 8, 10, 12, 14, 16$, using element-by-element operations.

A.13 Use MATLAB to show that the sum of the infinite series $\sum_{n=0}^{\infty} (-1)^n \frac{1}{(2n+1)}$ converges to $\pi/4$. Do it by computing the sum for:

- (a) $n = 100$
 (b) $n = 1,000$
 (c) $n = 5,000$

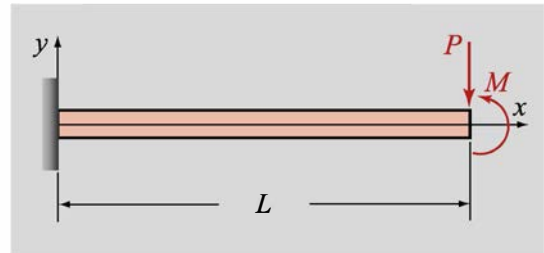
In each part create a vector n in which the first element is 0, the increment is 1, and the last term is either 100, 1,000, or 5,000. Then, use element-by-element calculation to create a vector in which the elements are $(-1)^n \frac{1}{(2n+1)}$. Finally, use the function `sum` to add the terms of the series. Compare the values obtained in parts *a*, *b*, and *c* with the value of $\frac{\pi}{4}$. (Do not forget to type semicolons at the end of commands that otherwise will display large vectors.)

A.14 A cantilever beam is loaded by a force $P = 850$ N and a moment $M = 3600$ N-m as shown. The deflection y at a point x along the beam is given by the equation:

$$y = \frac{1}{EI} \left[\frac{P}{6} (x^3 - 3Lx^2) + \frac{M}{2} x^2 \right]$$

where E is the elastic modulus, I is the moment of inertia, and L is the length of the beam. For the beam shown in the figure $L = 6$ m, $E = 70 \times 10^9$ Pa (aluminum), and $I = 9.19 \times 10^{-6}$ m⁴.

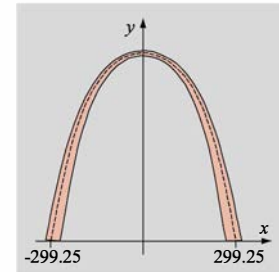
Plot the deflection of the beam y as a function of x .



A.15 The Gateway Arch in St. Louis is shaped according to the equation:

$$y = 693.8 - 68.8 \cosh\left(\frac{x}{99.7}\right) \text{ ft.}$$

Make a plot of the Arch for $-299.25 \leq x \leq 299.25$ ft.



A.16 Plot the function $f(x) = \frac{0.5x^3 - x^2}{x^2 - x - 20}$ for $-15 \leq x \leq 15$. Notice that the function has two vertical asymptotes. Plot the function by dividing the domain of x into three parts: one from -15 to near the left asymptote, one between the two asymptotes, and one from near the right asymptote to 15 . Set the range of the y -axis from -20 to 20 .

A.17 Write an anonymous MATLAB function for the following math function:

$$y(x) = xe^{-0.7x} \sqrt{2x^2 + 1}$$

The input to the function is x and the output is y . Write the function such that x can be a vector.

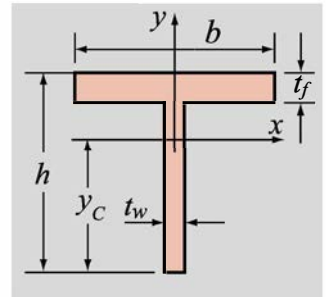
- (a) Use the function to calculate $y(3)$, and $y(8)$.
 (b) Use the function to make a plot of the function $y(x)$ for $0 \leq x \leq 10$.

A.18 The fuel efficiency of automobiles is measured in mi/Gal (miles per gallon) or in km/L (kilometers per liter). Write a MATLAB user-defined function that converts fuel efficiency values from mi/Gal (U.S. Gallons) to km/L. For the function name and arguments use `kmL=mgToKm(mpg)`. The input argument

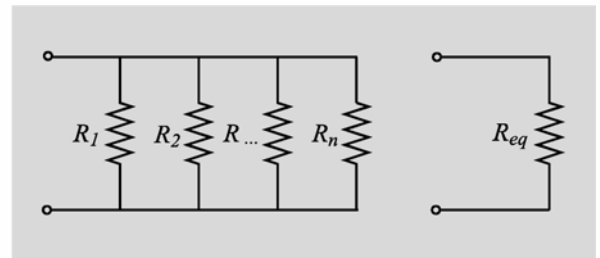
mpg is the efficiency in mi/Gal, and the output argument kmL is the efficiency in km/L. Use the function in the Command Window to to:

- Determine the fuel efficiency in km/L of a car that consumes 23 mi/Gal.
- Determine the fuel efficiency in km/L of a car that consumes 50 mi/Gal.

A.19 Write a user-defined MATLAB function that determines the cross-sectional area, A , the location of the centroid (the distance y_C), and moments of inertia I_{xx} and I_{yy} of an “T” beam. For the function name and arguments use $[A, y_C, I_{xx}, I_{yy}] = \text{MofIner}(b, h, t_f, t_w)$. The input arguments are the width, b , height, h , flange thickness, t_f , and web thickness, t_w , of the beam, in millimeters, as shown in the figure. The output arguments are the cross-sectional area (in mm^2), the location of the centroid (in mm), and the moments of inertia (in mm^4). Use the function in the Command Window to determine the cross-sectional area, centroid location, and moments of inertia and of a beam with $b = 300 \text{ mm}$, $h = 400 \text{ mm}$, $t_f = 20 \text{ mm}$, and $t_w = 12 \text{ mm}$.



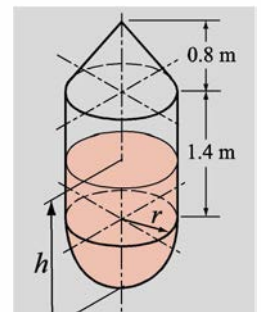
A.20 Write a user-defined MATLAB function that calculates the equivalent resistance, R_{eq} of n resistors R_1, R_2, \dots, R_n connected in parallel. For function name and arguments use $R_{eq} = \text{EqResistance}(R)$. The input argument is a vector whose elements are the values of the resistors. The output argument is the value of the equivalent resistance. The function should work for any number of resistors. Use the function in the Command Window to determine the equivalent resistance of the following five resistors that are connected in parallel $R_1 = 200\Omega$, $R_2 = 600\Omega$, $R_3 = 1000\Omega$, $R_4 = 100\Omega$, and $R_5 = 500\Omega$.



A.21 A vector is given by $x = [15 \ 85 \ 72 \ 59 \ 100 \ 80 \ 44 \ 60 \ 91 \ 38]$. Using conditional statements and loops write a program that determines the average of the elements of the vector that are larger than 59.

A.22 Write a user-defined function that creates a vector whose elements are the prime numbers between two numbers. For the function name and arguments use $\text{pr} = \text{Primary}(a, b)$. The input to the function are two numbers (integers) a and b (such that $a < b$), and the output pr is a vector in which the elements are the prime numbers between a and b .

A.23 Write a user-defined function that sorts the elements of a vector (of any length) from the largest to the smallest. For the function name and arguments use $y = \text{downsort}(x)$. The input to the function is a vector x of any length, and the output y is a vector in which the elements of x are arranged in descending order. Do not use the MATLAB `sort` function. Test your function by using it in the Command Window to rearrange the elements of the following vector: $[-2, 8, 29, 0, 3, -17, -1, 54, 15, -10, 32]$.



A.24 A cylindrical, vertical fuel tank has hemispheric end cap at the bottom and a conic end cap at the top as shown. The radius of the cylinder and the hemispheric end cap is $r = 60$ cm.

Write a user-defined function (for the function name and arguments use $V = V_{\text{fuel}}(h)$) that gives the volume of the fuel in the tank as a function of the height h . Use the function to make a plot of the volume as a function of h for $0 \leq h \leq 2.8$ m.

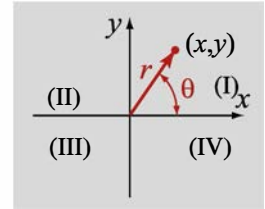
A.25 Write a user-defined MATLAB function that calculates the determinant of a 3×3 matrix by using the formula:

$$\det = A_{11} \begin{vmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{vmatrix} - A_{12} \begin{vmatrix} A_{21} & A_{23} \\ A_{31} & A_{33} \end{vmatrix} + A_{13} \begin{vmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{vmatrix}$$

For the function name and arguments use $d3 = \text{det3by3}(A)$, where the input argument A is the matrix and the output argument $d3$ is the value of the determinant. Write the code of det3by3 such that it has a subfunction that calculates the 2×2 determinant. Use det3by3 for calculating the determinants of:

$$(a) \begin{bmatrix} 1 & 3 & 2 \\ 6 & 5 & 4 \\ 7 & 8 & 9 \end{bmatrix}, \quad (b) \begin{bmatrix} -2.5 & 7 & 1 \\ 5 & -3 & -2.6 \\ 4 & 2 & -1 \end{bmatrix}.$$

A.26 Write a user-defined function that determines the polar coordinates of a point from the Cartesian coordinates in a two-dimensional plane. For the function name and arguments use $[\text{radius } \theta] = \text{CartToPolar}(x, y)$. The input arguments are the x and y coordinates of the point, and the output arguments are the radial distance to the point and angle θ . The angle θ is in degrees and is measured relative to the positive x axis, such that it is a number between 0 and 90 in quadrant I, between 90 and 180 in quadrant II, between 180 and 270 in quadrant III, and between 270 and 360 in quadrant IV. Use the function to determine the polar coordinates of points $(-15, -3)$, $(7, 12)$, $(17, -9)$, and $(-10, 6.5)$.



A.27 Write a user-defined function that adds two vectors that are written in polar coordinates. For the function name and arguments use $[\text{vapb}] = \text{VecAddPolar}(va, vb)$. The input arguments va and vb are the vectors to be added. The output argument $vapb$ is the results. Each is a two-element MATLAB vector where the first element is the angle (between 0 and 360 in degrees) and the second element is the magnitude of the radius. To add the vectors the program first converts each of the vectors to Cartesian coordinates, then adds the vectors, and finally converts the result to polar coordinates and assigns it to the output argument $vapb$. The code is written such that it includes subfunctions for the conversion of the input vectors to Cartesian coordinates, and for the conversion of the result to polar coordinates (the user-defined function CartToPolar from Problem A.26 can be used for the latter task). Use the function in the Command Window to:

- (a) Add the vectors $a = [32^\circ, 50]$ and $b = [60^\circ, 70]$.
- (b) Add the vectors $c = [167^\circ, 58]$ and $d = [250^\circ, 90]$.
- (c) Add the vectors $e = [25^\circ, 43]$ and $g = [290^\circ, 115]$.