Lecture Notes in Advanced Computer Skills (MATLAB) for MSc Degree of Mathematics

Imad A. Aziz

Erbil

2022-2023

INTRODUCTION to MATLAB

1.1 INTRODUCTION

MATLAB, which stands for **MAT**rix **LAB**oratory, is a state-of-the-art mathematical software package, which is used extensively in both academia and industry. It is an interactive program for numerical computation and data visualization, which along with its programming capabilities provides a very useful tool for almost all areas of science and engineering. Unlike other mathematical packages, such as MAPLE or MATHEMATICA, MATLAB cannot perform symbolic manipulations without the use of additional Toolboxes. It remains however, one of the leading software packages for numerical computation

1.1.1 Starting MATLAB

However you start MATLAB, you will briefly see a window that displays the MATLAB logo as well as some product information, and then a MATLAB Desktop window will launch. That window will contain a title bar, a menu bar, a tool bar and four embedded windows. The largest and most important window is the Command Window on the middle, the Command History Window and the Workspace in right, the Current Directory Browser in left. For now we concentrate on the Command Window in order to get you started issuing MATLAB commands as quickly as possible. At the top of the Command Window, you may see some general information about MATLAB, perhaps some special instructions for getting started or accessing help, but most important of all, you will see a command prompt (>>). If the Command Window is "active," its title bar will be dark, and the prompt will be followed by a cursor (a blinking vertical line). That is the place where you will enter your MATLAB commands. If the Command Window is not active, just click in it anywhere. Figure 1.1 contains an example of a newly launched MATLAB Desktop.

Figure 1.1

a. **Command Window: -** Use the Command Window to enter variables and to run MATLAB functions and scripts. MATLAB displays the results. Press the up arrow key ↑ to recall a statement you previously typed. Edit the statement as needed, and then press Enter to run it.

b. **Command History:-**Statements you enter in the Command Window are logged with a timestamp in the Command History. From the Command History, you can view and search for previously run statements, as well as copy and execute selected statements.

c. **Workspace: -** The workspace consists of the set of variables built up during a session of using the MATLAB software and stored in memory. You add variables to the workspace by using functions, running M-files, and loading saved workspaces**.**

d. **Current Directory Browser:-**the files and subdirectories it contains are listed in the Current Directory Browser.

1.1.2 Typing in Command Window

Abort
In order to *abort* a command in MATLAB, hold down the control key and press c to generate a local abort with MATLAB.

The Semicolon (;)
If a semicolon (;) is typed at the end of a command the output of the command is not displayed.

Typing %
When percent symbol (%) is typed in the beginning of a line, the line is designated as a comment. When the *enter* key is pressed the line is not executed.

The clc Command
Typing *clc* command and pressing *enter* cleans the command window. Once the *clc* command is executed a clear window is displayed.

Help
MATLAB has a host of built-in functions. For a complete list, refer to MATLAB users
guide or refer to the *on line Help*. To obtain help on a particular topic in the list, *e.g.*, inverse, type *help inv*.

1.1.3 Display Format

MATLAB has several different screen output formats for displaying numbers. These formats can be found by typing the help command: help format in the Command Window. A few of these formats are shown in Table

`Style`	Result	Example
`short`	Short, fixed-decimal format with 4 digits after the decimal point. This is the default numeric setting.	`3.1416`

`Style`	Result	Example
long	Long, fixed-decimal format with 15 digits after the decimal point for double values, and 7 digits after the decimal point for single values.	3.141592653589793
shortE	Short scientific notation with 4 digits after the decimal point.	3.1416e+00
longE	Long scientific notation with 15 digits after the decimal point for double values, and 7 digits after the decimal point for single values.	3.141592653589793e+00
shortG	Short, fixed-decimal format or scientific notation, whichever is more compact, with a total of 5 digits.	3.1416
longG	Long, fixed-decimal format or scientific notation, whichever is more compact, with a total of 15 digits for double values, and 7 digits for single values.	3.14159265358979
shortEng	Short engineering notation (exponent is a multiple of 3) with 4 digits after the decimal point.	3.1416e+000
longEng	Long engineering notation (exponent is a multiple of 3) with 15 significant digits.	3.14159265358979e+000
bank	Currency format with 2 digits after the decimal point.	3.14
hex	Hexadecimal representation of a binary double-precision number.	400921fb54442d18
rational	Ratio of small integers.	355/113

1.2 Arithmetic Operations

The symbols for arithmetic operations with scalars are summarized below in Table

Symbol	Operation	Example
+	Addition	6+3=9
-	Subtraction	6-3=3
*	Multiplication	6*3=18
\	Left Division	6\3=1/2
/	Right Division	6/3=2
^	Power	6^3=216

1.3 Elementary Math Built in Functions

MATLAB contains a number of functions for performing computations which require the use of logarithms, elementary math functions, and trigonometric math functions. List of these commonly used elementary MATLAB mathematical built-in functions are given in Tables.

Modulo Division and Rounding

Function	Description
mod	Remainder after division (modulo operation)
rem	Remainder after division
ceil	Round toward positive infinity
fix	Round toward zero
floor	Round toward negative infinity
round	Round to nearest decimal or integer

Exponents and Logarithms

Function	Description
exp	Exponential
log	Natural logarithm
log10	Common logarithm (base 10)
log2	Base 2 logarithm and floating-point number dissection
sqrt	Square root

Trigonometry

Sine

Function	Description
sin	Sine of argument in radians
sind	Sine of argument in degrees
sinpi	Compute sin(X*pi) accurately
asin	Inverse sine in radians
asind	Inverse sine in degrees
sinh	Hyperbolic sine
asinh	Inverse hyperbolic sine

Cosine

Function	Description
cos	Cosine of argument in radians
cosd	Cosine of argument in degrees
cospi	Compute cos(X*pi) accurately
acos	Inverse cosine in radians
acosd	Inverse cosine in degrees
cosh	Hyperbolic cosine
acosh	Inverse hyperbolic cosine

Tangent

Function	Description
tan	Tangent of argument in radians
tand	Tangent of argument in degrees
atan	Inverse tangent in radians
atand	Inverse tangent in degrees
atan2	Four-quadrant inverse tangent
atan2d	Four-quadrant inverse tangent in degrees
tanh	Hyperbolic tangent
atanh	Inverse hyperbolic tangent

1.4 Variable Names

A variable is a name made of a letter or a combination of several letters and digits. Variable names can be up to 63 (in MATLAB 7) characters long (31 characters on MATLAB 6.0). MATLAB is case sensitive. For instance, XX, Xx, xX, and xx are the names of four different variables. It should be noted here that not to use the names of a built-in functions for a variable. For instance, avoid using: sin, cos, exp, sqrt, ..., etc. Once a function name is used to define a variable, the function cannot be used.

1.4.1 Predefined Variables

MATLAB includes a number of predefined variables. Some of the predefined variables that are available to use in MATLAB programs are summarized in Table

Expression	Description
`pi`	The number π up to 15 significant digits.
`i, j`	The complex number
`inf`	Represents the mathematical Infinity concept, for example, a result of division by zero.
`NaN`	Stands for Not-A-Number. Represents the result of a meaningless mathematical function, like $0/0$.
`clock`	Contains the current date and time in the form of a 6-element row vector: year, month, day, hour, minute, second.
`date`	Contains a string representing today's date.
`eps`	Stands for **epsilon**. It represents the smallest number that can be represented by your MATLAB software.
`ans`	A special variable that MATLAB uses to store the result of MATLAB's command line.

1.4.2 Command for Managing Variables

Table below lists commands that can be used to eliminate variables or to obtain information about variables that have been created. The procedure is to enter the command in the Command Window and the Enter key is to be pressed.

Expression	Description
`clear`	Remove items from workspace, freeing up system memory
`clc`	Clear Command Window
`who`	List variables in workspace
`whos`	List variables in workspace, with sizes and types

1.5 Complex Numbers

Complex numbers consist of two separate parts: a real part and an imaginary part. The basic imaginary unit is equal to the square root of -1. This is represented in MATLAB by either of two letters: i or j.

1.5.1 Creating Complex Numbers

The following statement shows one way of creating a complex value in MATLAB. The variable x is assigned a complex number with a real part of 2 and an imaginary part of 3:

x = 2 + 3i;

Another way to create a complex number is using the complex function. This function combines two numeric inputs into a complex output, making the first input real and the second imaginary:

```
x = 4;
y = -1;
z = complex(x, y)
z =
   4.0000 -1.0000i
```

1.5.2 Arithmetic operations with complex numbers

Symbol	Operation	Example
+	Addition	z = x + y
-	Subtraction	z = x - y
*	Multiplication	z = x * y
\	Left Division	z = x / y

1.5.3 Complex number functions

Function	Description
abs	Absolute value and complex magnitude
angle	Phase angle
complex	Create complex array
conj	Complex conjugate
i	Imaginary unit
imag	Imaginary part of complex number
isreal	Determine whether array uses complex storage
j	Imaginary unit
real	Real part of complex number

VECTORS AND MATRICES

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 – 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]

2.2 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,

$$A = \begin{bmatrix} 1 & 3 & -4 \\ 0 & -2 & 8 \end{bmatrix}$$

The MATLAB input command is

A = [1 3 – 4 ; 0 – 2 8]

Similarly, for complex number elements of a matrix B

$$B = \begin{bmatrix} -5x & \ln 2x + 7\sin 3y \\ 3i & 5 - 13i \end{bmatrix}$$

The MATLAB input command is

B = [– 5 * x log(2 * x) + 7 * sin (3 * y) ; 3i 5 – 13i]

2.3 Addressing Arrays

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

2.3.1 Colon for a vector

Va(:) – refers to all the elements of the vector Va (either a row or a column vector).
Va(m : n) – refers to elements m through n of the vector Va.
For instance

```
>> V = [2 5 – 1 11 8 4 7 – 3 11]
>> u = V(2 : 8)
u = 5 – 1 11 8 4 7 – 3 11
```

2.3.2 Colon for a matrix

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

- A(:,n) is the nth column of matrix A.
- A(m,:) is the mth row of matrix A.
- A(:,:,p) is the pth page of three-dimensional array A.
- A(:) reshapes all elements of A into a single column vector. This has no effect if A is already a column vector.
- A(:,:) reshapes all elements of A into a two-dimensional matrix. This has no effect if A is already a matrix or vector.

- A(j:k) uses the vector j:k to index into A and is therefore equivalent to the vector [A(j), A(j+1), ..., A(k)].
- A(:,j:k) includes all subscripts in the first dimension but uses the vector j:k to index in the second dimension. This returns a matrix with columns [A(:,j), A(:,j+1), ..., A(:,k)].

2.3.3 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.

```
>> A = [1 2 3;4 5 6;7 8 10]
A =
        1 2 3
        4 5 6
        7 8 10


>> A(:, 2) = []
A =
        1 3
        4 6
        7 10
```

2.3.4 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.

```
>> A = [A(:,1) [2 5 8]’ A(:,2)]
A =
        1 2 3
        4 5 6
        7 8 10
```

2.4 Element-by-element operations

Element-by-element operations can only be done with arrays of the same size. Elementby- 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 below lists these operations

Operator	Purpose	Description	Reference Page
+	Addition	`A+B` adds `A` and `B`.	`plus`
+	Unary plus	`+A` returns `A`.	`uplus`
-	Subtraction	`A-B` subtracts `B` from `A`	`minus`
-	Unary minus	`-A` negates the elements of `A`.	`uminus`
.*	Element-wise multiplication	`A.*B` is the element-by-element product of `A` and `B`.	`times`
.^	Element-wise power	`A.^B` is the matrix with elements `A(i,j)` to the `B(i,j)` power.	`power`
./	Right array division	`A./B` is the matrix with elements `A(i,j)/B(i,j)`.	`rdivide`
.\	Left array division	`A.\B` is the matrix with elements `B(i,j)/A(i,j)`.	`ldivide`
.'	Array transpose	`A.'` is the array transpose of `A`. For complex matrices, this does not involve conjugation.	`transpose`

Matrix Operations

Matrix operations follow the rules of linear algebra and are not compatible with multidimensional arrays. The required size and shape of the inputs in relation to one another depends on the operation. For non-scalar inputs, the matrix operators generally calculate different answers than their array operator counterparts.

The following table provides a summary of matrix arithmetic operators in MATLAB.

Operator	Purpose	Description	Reference Page
`*`	Matrix multiplication	`C = A*B` is the linear algebraic product of the matrices `A` and `B`. The number of columns of `A` must equal the number of rows of `B`.	`mtimes`
`\`	Matrix left division	`x = A\B` is the solution to the equation *Ax* = *B*. Matrices `A` and `B` must have the same number of rows.	`mldivide`
`/`	Matrix right division	`x = B/A` is the solution to the equation *xA* = *B*. Matrices `A` and `B` must have the same number of columns. In terms of the left division operator, `B/A = (A'\B')'`.	`mrdivide`
`^`	Matrix power	`A^B` is `A` to the power `B`, if `B` is a scalar. For other values of `B`, the calculation involves eigenvalues and eigenvectors.	`mpower`
`'`	Complex conjugate transpose	`A'` is the linear algebraic transpose of `A`. For complex matrices, this is the complex conjugate transpose.	`ctranspose`

2.5 Identity, Ones and Zeros Matrix

The function *eye(m,n)*, ones(m,n) , zeros(m,n) returns an m-by-n rectangular identity ,all elements ones, all elements zero matrix and eye(n) returns an n-by-n square identity matrix and same for others.

2.6 Built-in Functions for Arrays

Table below lists some of the many built-in functions available in MATLAB for analyzing
arrays.

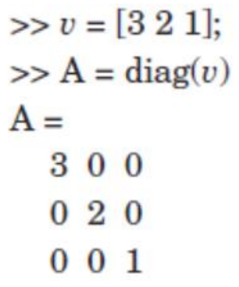
a. Transpose

In MATLAB, the transpose of the matrix A is denoted by A'

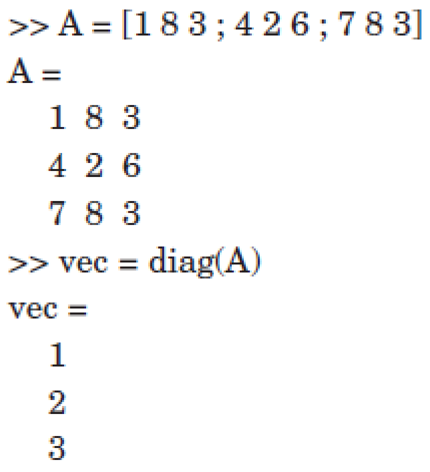
b. Diagonal of matrix

diag (A) :-When A is a matrix, creates a vector from the diagonal elements of A.

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
```



```
>> A = [1 8 3 ; 4 2 6 ; 7 8 3]
A =
   1  8  3
   4  2  6
   7  8  3
>> vec = diag(A)
vec =
   1
   2
   3
```

2.7 Matrix dimension functions

- **length(A)**: - Length of vector or largest array dimension.
- **size(A)**: - returns the sizes of each dimension of array.
- **ndims(A)**: - Number of array dimensions.
- **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*.

Example: - Let

$$A = \begin{pmatrix} 2 & 5 & 0 & 7 \\ 3 & 4 & -1 & 4 \end{pmatrix}$$

Then

```
>> length(A)
ans =
    4
>> size(A)
ans =
    2     4
>> ndims(A)
ans =
    2
>> reshape(A,4,2)
```

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

2.8 Relational and Logical Operators

A relational operator compares two numbers by finding whether a comparison statement is true (1) or false (0). A logical operator examines true/false statements and produces a result which is true or false according to the specific operator.

Relational operators Table

Symbol	Function Equivalent	Description
`<`	`lt`	Less than
`<=`	`le`	Less than or equal to
`>`	`gt`	Greater than
`>=`	`ge`	Greater than or equal to
`==`	`eq`	Equal to
`~=`	`ne`	Not equal to

Logical operators Table

Symbol	Role	More Information
&	Find logical AND	`and`
|	Find logical OR	`or`
&&	Find logical AND (with short-circuiting)	`Short-Circuit AND`
||	Find logical OR (with short-circuiting)	`Short-Circuit OR`
~	Find logical NOT	`not`

Additional logical built-in functions Table

Function	Description
`xor`	Find logical exclusive-OR
`all`	Determine if all array elements are nonzero or `true`
`any`	Determine if any array elements are nonzero
`find`	Find indices and values of nonzero elements
`islogical`	Determine if input is logical array
`logical`	Convert numeric values to logical
`true`	Logical 1 (true)
false	Logical 0 (false)

PROGRAMMING IN MATLAB

3.1 M-files: Scripts and functions

To take advantage of MATLAB's full capabilities, we need to know how to construct long (and sometimes complex) sequences of statements. This can be done by writing the commands in a file and calling it from within MATLAB. Such files are called "m-files" because they must have the filename extension ".m". This extension is required in order for these files to be interpreted by MATLAB.

There are two types of m-files: *script files* and *function files*.

Script files contain a sequence of usual MATLAB commands, that are executed (in order) once the script is called within MATLAB. For example, if such a file has the name compute .m , then typing the command compute at the MATLAB prompt will cause the statements in that file to be executed. Script files can be very useful when entering data into a matrix.

Example:- Create the script file then write a program to find the roots of equation $2x^2$ -3x+1=0.

Sol:

```
% This Program written to Find the Roots of Equation 2x^2-
3x+1=0%
 a=2; b=-3; c=1;
 x1=(-b+sqrt(b^2-4*a*c))/2
 x2=(-b-sqrt(b^2-4*a*c))/2
```

3.2 Input and output command

a. *Disp(X)*

disp(X) displays an array, without printing the array name. If X contains a text string, the string is displayed. Another way to display an array on the screen is to type its name, but this prints a leading "X=," which is not always desirable. Note that *disp* does not display empty arrays.

```
>> disp('string expression');
```

```
string expression
>> A=[3,2;2,3];
disp(A);
    3     2
    2     3
```

b. Input

The input function can be used for requesting user input. For example,

```
r=input('value for r: ');
```

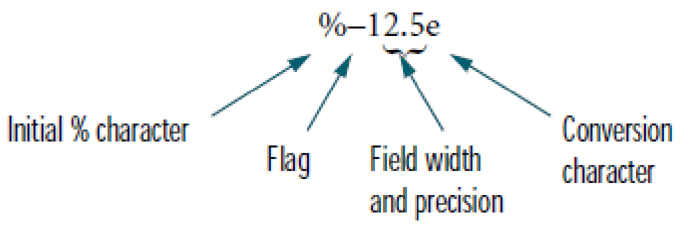
displays `value for r:`

to the screen and waits for the user to enter an expression which is then assigned to r.

c. fprintf

The *fprintf* command displays output (text and data) on the screen or saves it to a file. The output can be formatted using this command.

fprintf (format,A,...) writes to standard output—the screen. The format string specifies notation, alignment, significant digits, field width, and other aspects of output format. It can contain ordinary alphanumeric characters; along with escape characters, conversion specifiers, and other characters, organized as shown below:



For more information, see “Tables” and “References”

Code	Conversion instruction
%s	format as a string
%d	format with no fractional part (integer format)
%f	format as a floating-point value
%e	format as a floating-point value in scientific notation
%g	format in the most compact form of either %f or %e
\n	insert newline in output string
\t	insert tab in output string

Example:-

```
 >> x = 2; y = sqrt(x);
 >> fprintf('The squrt root of %g is %9.4f\n',x,y)
        The squrt root of 2 is    1.4142
```

Example:-

```
>> x = 1:4; y = sqrt(x);
>> fprintf('The squrt root of %d is %4.2f\n',[x;y])
The squrt root of 1 is 1.00
The squrt root of 2 is 1.41
The squrt root of 3 is 1.73
The squrt root of 4 is 2.00
```

3.3 Conditional Execution

3.3.1 Conditional Execution or Branching:

As the result of a comparison, or another logical (true/false) test, selected blocks of program code are executed or skipped. Conditional execution is implemented with if, if...else, and if...elseif constructs, or with a switch construct.

There are three types of if constructs

1. Plain if
2. if...else
3. if...elseif

if Constructs
Syntax:

```
 if expression
         block of statements
 end
```

The *block of statements* is executed only if the *expression* is true.

Example:

```
  if a < 0
    disp(’a is negative’);
  end
```

One-line format uses comma after if *expression*

```
if a < 0, disp(’a is negative’); end
```

if. . . else

Multiple choices are allowed with **if. . . else** and **if. . . elseif** constructs

```
if x < 0
```

```
  error('x is negative; sqrt(x) is imaginary');
else
  r = sqrt(x);
end
```

if. . . elseif
It's a good idea to include a default **else** to catch cases that don't match preceding **if** and **elseif** blocks

```
if x > 0
   disp('x is positive');
elseif x < 0
   disp('x is negative');
else
   disp('x is exactly zero');
end
```

3.3.2 The switch Construct

A switch construct is useful when a test value can take on discrete values that are either integers or strings.

Syntax:

```
switch expression
  case value1,
    block of statements
  case value2,
    block of statements
     ...
 otherwise,
   block of statements
end
```

Example:

```
color = ’...’; % color is a string
switch color
 case ’red’
   disp(’Color is red’);
 case ’blue’
   disp(’Color is blue’);
 case ’green’
   disp(’Color is green’);
 otherwise
  disp(’Color is not red, blue, or green’);
end
```

3.4 Repetition or Looping

A sequence of calculations is repeated until either
1. All elements in a vector or matrix have been processed
or
2. The calculations have produced a result that meets a predetermined termination criterion
Looping is achieved with for loops and while loops.

3.4.1 for loops

for loops are most often used when each element in a vector or matrix is to be processed.

Syntax:
```
for index = expression
  block of statements
end
```
Example: Sum of elements in a vector

```
x = 1:5; % create a row vector
sumx = 0; % initialize the sum
for k = 1:length(x)
  sumx = sumx + x(k);
end
```

3.4.2 for loop variations

Example: A loop with an index incremented by two

```
for k = 1:2:n
  ...
end
```
Example: A loop with an index that counts down

```
for k = n:-1:1
  ...
end
```
Example: A loop with non-integer increments

```
for x = 0:pi/15:pi
 fprintf('%8.2f %8.5f\n',x,sin(x));
end
```
Note: In the last example, x is a scalar inside the loop. Each time through the loop, x is set equal to one of the columns of 0:pi/15:pi.

3.4.3 while loops

while loops are most often used when an iteration is repeated until some termination criterion is met.

Syntax:

```
while expression
  block of statements
end
```

The block of statements is executed as long as expression is true.

Example: Here is a simple example of a script M-file that uses while to numerically sum the infinite series 1/14 + 1/24 + 1/34 +· · ·, stopping only when the terms become so small (compared to the machine precision) that the numerical sum stops changing:

```
n = 1; oldsum = -1; newsum = 0;
while newsum > oldsum
  oldsum = newsum;
  newsum = newsum + nˆ(-4);
  n = n + 1;
end
newsum
```

Note:-It is (almost) always a good idea to put a limit on the number of iterations to be performed by a while loop.
An improvement on the preceding loop,

```
 n = 1; oldsum = -1; newsum = 0;
 maxit = 25; it = 0;
  while newsum > oldsum & it<maxit
    oldsum = newsum;
    newsum = newsum + nˆ(-4);
    n = n + 1;
```

```
      it = it + 1;
   end
 newsum
```

4.1 Function files

on the other hand, play the role of user defined commands that often have input and output. You can create your own commands for specific problems this way, which will have the same status as other MATLAB commands. Let us give a simple example. The text below is saved in a file called log3.m and it is used to calculate the base 3 logarithm of a positive number. The text file can be created in a variety of ways, for example using the built-in MATLAB editor through the command edit (that is available with MATLAB 5.0 and above), or your favorite (external) text editor (e.g. *Notepad* or *Wordpad* in Microsoft Windows). You must make sure that the filename has the extension ".m" !

```
function [a] = log3(x)
a = log(abs(x))./log(3);
end
» log3(5)
ans =
      1.4650
```

Syntax:

The first line of a function m-file has the form:

```
    function [outArgs] = funName(inArgs)
```

outArgs are enclosed in [ ]

- outArgs is a comma-separated list of variable names
- [ ] is optional if there is only one parameter
- functions with no outArgs are legal

inArgs are enclosed in ( )

- inArgs is a comma-separated list of variable names
- functions with no inArgs are legal

Example:- Write the function to find Fibonnaci sequence.

```
function f = Fib1(n)
F=zeros(1,n+1);
F(2) = 1;
for i = 3:n+1
F(i) = F(i-1) + F(i-2);
End
End
```

Example:- Write the function to find area and perimeter of triangle.

```
function [A s] = area(a,b,c)
s = (a+b+c)/2;
A = sqrt(s*(s-a)*(s-b)*(s-c));
End
```

4.2 Loop and Function Cotroal

a. return

return is used to force an exit from a function. This can have the effect of escaping from a loop. Any statements following the loop that are in the function body are skipped.

b. break

break is used to escape from an enclosing while or for loop. Execution continues at the end of the enclosing loop construct.

Example: - write a function to check that the input number is prime or not.

```
function [p] = prim(x)
p=0;
for i=2:x-1
    if rem(x,i)==0
```

```
        p=1;
        return
    end
end
end
```

in above example we have only one element input but if we write program to work with array of element as an input we must write the program as follow

```
function p = prim(a)
p=zero(size(a));
for i=1:length(a)
  for j=2:a(i)-1
    if rem(a(i),j)==0
        p(i)=1;
        Break
    end
  end
end
end
```

c. continue

The continue statement passes control to the next iteration of the for loop or while loop in which it appears, skipping any remaining statements in the body of the loop. In nested loops, continue passes control to the next iteration of the for loop or while loop enclosing it.

Example:- find the s where

$$s = \sum_{\substack{i=0 \\ i\neq 3}}^{n} \frac{1}{i-3}$$

```
n=input('n=')
s=0;
for i=0:n
    if i==3, continue, end
    s=s+1/(i-3);
```

```
end
s
```

d. error (‘*text*’)

Terminates execution and displays the message contained in text on the screen.

Example: - Write the function to replace the last row with last column of input matrix.

```
 function b = lr2lc(a)
  [n m]=size(a);
  if n~=m
      error('The matrix must be square')
  else
      b=a;
      b(n,:)=a(:,m);
      b(:,m)=a(n,:);
  end
  end
```

STRINGS

5.1 Character Strings

While Matlab is primarily intended for number crunching, there are times when it is desirable to manipulate text, as is needed in placing labels and titles on plots. In Matlab, text is referred to as character strings or simply strings.

5.1.1 String Construction

Character strings in Matlab are special numerical arrays of ASCII values that are displayed as their character string representation. For example:

```
>> text = ’This is a character string’
   text =
          This is a character string
>> size(text)
   ans =
         1 26
>> whos
   Name    Size    Bytes    Class
   ans     1x2      16      double array
   text    1x26     52      char array
```

Grand total is 28 elements using 68 bytes

5.1.2 ASCII Codes

Each character in a string is one element in an array that requires two bytes per character for storage, using the ASCII code. This differs from the eight bytes per element required for numerical or double arrays. The ASCII codes for the letters 'A' to 'Z' are the consecutive integers from 65 to 90, while the codes for 'a' to 'z' are 97 to 122. The function abs returns the ASCII codes for a string.

```
>> text = ’This is a character string’
text =
     This is a character string
>> d = abs(text)
 d =
  Columns 1 through 12

    84  104  105  115  32  105  115  32  97  32  99

104

  Columns 13 through 24

    97  114  97  99  116  101  114  32  115  116  114

105

  Columns 25 through 26

    110  103
```

The function **char** performs the inverse transformation, from ASCII codes to a string:

```
>> char(d)
ans =
    is a character string
```

The relationship between strings and their ASCII codes allow you to do the following:

```
>> alpha = abs('a'):abs('z');

>> disp(char(alpha))

    Abcdefghijklmnopqrstuvwxyz
```

5.2 Strings are Arrays

Since strings are arrays, they can be manipulated with array manipulation tools:

```
>> text = 'This is a character string';
>> u = text(11:19)
u =
   character
```

As with matrices, character strings can have multiple rows, but each row must have an equal number of columns. Therefore, blanks are explicitly required to make all rows the same length.
For example:

```
>> v = [’Character strings having more than’
        ’one row must have the same number ’
        ’of columns - just like matrices ’]
v =
   Character strings having more than
   one row must have the same number
   of columns - just like matrices

>> size(v)
    ans =
         3 34
```

5.3 Concatenation of Strings

Because strings are arrays, they may be concatenated (joined) with square brackets. For example:

```
>> today = ’May’;
>> today = [today, ’ 18’]
   today =
          May 18
```

5.4 String Conversions

num2str(*x*) :- Converts the matrix ***x*** into a string representation with about 4 digits and an exponent if required.

The num2str function can be used to convert numerical results into strings for use in formatting displayed results with disp. For example;

```
a=input('a=');
for i=1:length(a);
    if isprime(a(i))
```

```
        disp([num2str(a(i)) ' is prime'])
    else
        disp([num2str(a(i)) ' is not prime'])
    end
end
```

run

```
 a=[2 8 5]
  2 is prime
  8 is not prime
  5 is prime
```

str2num(*s*) :- converts the string ***s*** which is an ASCII character representation of a numeric value, to numeric representation. **str2num** also converts string matrices to numeric matrices. If the input string does not represent a valid number or matrix, str2num(s) returns the empty matrix. For example;

```
 s='423'  »  str2num(s)=423

  s='3 0 2 5' »  a=str2num(s)
  a= 3 0 2 5 is matrix

  str2num('2 4; 6 8')

    ans =
            2     4
            6     8
```

eval(*s*):- Execute the string s as a Matlab expression or statement.

Example:- write a program to draw the graph of input function.

```
f = input( ’Enter function (of x) to be plotted: ’,
’s’);
x = 0:0.01:10;
```

```
y= eval(f);

plot(x,y),grid
```

Here's how the command line looks after you enter the function:

```
>> Enter function (of x) to be plotted:
    exp(-0.5*x) .* sin(x)
```

Example:- You can concatenate strings to create a complete expression for input to eval. This code shows how eval can create 10 variables named P1, P2, ...P10, and set each of them to a different value.

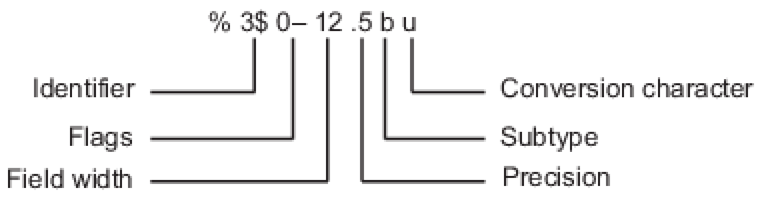
```
for i=1:10

  eval(['P',int2str(i),'= i.^2'])

 end
```

fprintf(formatspec,var)

formats var as specified in formatSpec.

Formatting Operator

A formatting operator starts with a percent sign, %, and ends with a conversion character. It is mandatory to specify the conversion character. Optionally, you can specify identifier, flags, field width, precision, and subtype operators between % and the conversion character. Spaces are invalid between operators and are shown here only for readability.



Conversion Character

This table shows conversion characters to format numeric and character data as text.

Value Type	Conversion	Details
Integer, signed	%d or %i	Base 10
Integer, unsigned	%u	Base 10
	%o	Base 8 (octal)
	%x	Base 16 (hexadecimal), lowercase letters a–f
	%X	Same as %x, uppercase letters A–F
Floating-point number	%f	Fixed-point notation (Use a precision operator to specify the number of digits after the decimal point.)
	%e	Exponential notation, such as 3.141593e+00 (Use a precision operator to specify the number of digits after the decimal point.)
	%E	Same as %e, but uppercase, such as 3.141593E+00 (Use a precision operator to specify the number of digits after the decimal point.)
	%g	The more compact of %e or %f, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
	%G	The more compact of %E or %f, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
Characters or strings	%c	Single character
	%s	Character vector or string array. The type of the output text is the same as the type of formatSpec.

5.5 Some necessary subjects

5.5.1 Function handles

A function handle (@) is a reference to a function that can then be treated as a variable. It can be copied, placed in cell array, and evaluated just like a regular function.
Function handles can refer to built-in MATLAB functions, to your own function in an M-file, or to anonymous functions. An anonymous function is defined with a one-line expression, rather than by an M-file. Try:

```
g = @(x) x^2-5*x+6-sin(9*x)
g(1)=1.5879
```

Some MATLAB functions that operate on function handles need to evaluate the function on a vector, so it is often better to define an anonymous function (or M-file)so that it can operate entry-wise on scalars, vectors, or
matrices. Try this instead:

```
g = @(x) x.^2-5*x+6-sin(9*x)
g([-1 0 2 3])
ans =
   12.4121    6.0000    0.7510   -0.9564
```

The general syntax for an anonymous function is

```
fname = @(var1, var2, ...) expression
```

Here is an example with two input arguments:

```
norm2 = @(x,y) sqrt(x^2 + y^2)
norm2(4, 5)
```

Example 2.9: - Write a program to draw the surface of input function on square region around origin point.

```
f=input('input handle f(x,y)=');
[x1,y1]=meshgrid(-3:0.1:3);
z1=f(x1,y1);
```

```
surf(x1,y1,z1)
```

should input the function as `@(x,y)x.^2+y.^2`

5.5.2 Cell arrays

Cell arrays are arrays which contain elements of arbitrary types. They are identified by curly braces instead of square ones:

```
>> c = {[3,4],18.2,[1,2;2,1],'string'};
```

defines a cell array c. We can access elements of c in the following way:

```
>> c{3}
ans =
     1 2
     2 1
```

Example 2.10: - You can substitute multiple symbolic expressions, numeric expressions, or any combination, using cell arrays of symbolic or numeric values. Try:

```
syms x y
S = x^y
subs(S, x, 3)
subs(S, {x y}, {3 2})
subs(S, {x y}, {3 x+1})
```

Example 2.11: - Let

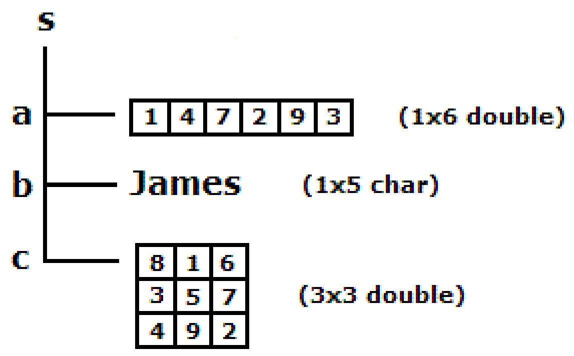
```
D = {‘red’;‘blue’;‘green’;‘yellow’}
```

```
 D =                                 sort(D)
 ‘red’                               ans =
 ‘blue’                                  'blue'
 ‘green’                                 'green'
 ‘yellow’                                'red'
                                         'yellow'
```

5.5.3 Structure

A structure is a MATLAB data type that provides the means to store hierarchical data together in a single entity. A structure consists mainly of data containers, called fields, and each of these fields stores an array of some MATLAB data type. You assign a name to each field as you create the structure. The figure below shows a structure s that has three fields: a, b, and c.

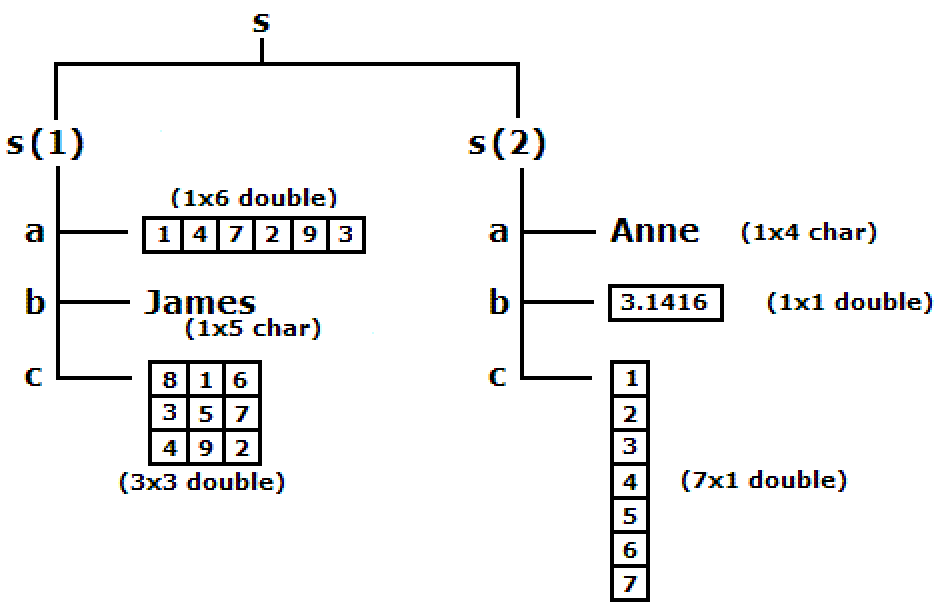
Example 2.12: -



```
s.a=[1 4 7 2 9 3];
s.b='James'
s.c=[8 1 6;3 5 7;4 9 2];
```

Like all MATLAB data types, the structure is an array. The class of a structure is called struct, so an array of structures is often referred to as a struct array. Like other MATLAB arrays, a struct array can have any dimensions. The struct array shown below has the dimensions 1-by-2 and is composed of two elements: s(1) and s(2). Each of these elements is a structure with fields a, b, and c of its own.

Example 2.13: -



```
s(1).a=[1 4 7 2 9 3];
s(1).b='James'
s(1).c=[8 1 6;3 5 7;4 9 2];
s(2).a='Anne';
s(2).b=pi;
s(2).c=[1;2;3;4;5;6;7];
```

Polynomials

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

$$f(x) = a_0x_n + a_1x_{n-1} + a_2x_{n-2} + \dots + a_{n-1}x_1 + a_n$$

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.

2.1 Input Polynomial

A vector represents a polynomial in MATLAB. When entering the data in MATLAB, simply enter each coefficient of the polynomial into the vector in descending order.

Example 2.1:- consider the polynomial $p(s) = 5s^5 + 7s^4 + 2s^2 - 6s + 10$

To enter this into MATLAB, we enter this as a vector as

```
>> p = [5 7 0 2 - 6 10]
p =
      5 7 0 2 - 6 10
```

It is necessary to enter the coefficients of all the terms.

2.2 Polynomial multiplication and division

MATLAB contains functions that perform polynomial multiplication and division, which are listed below:

conv(*p*, *q*)

Let $p(x)$ and $q(x)$ be two polynomial of degree n and m respectively, to find $h(x) = p(x) * q(x)$ using **h=conv(*p*, *q*)** computes a coefficient vector that contains the coefficients of the product of polynomials represented by the coefficients in *p* and *q*. The vectors *p* and *q* do not have to be the same size.

Example 2.2:- consider the polynomials
$p(x) = x^4 - 4x^2 + 2x + 1$ and $q(x) = x + 2$
then to find $h(x) = p(x) * q(x)$ write Matlab code as follows

```
p=[1 0 -4 2 1];
q=[1 2];
h=conv(p,q)
```

result is

```
h =
     1     2    -4    -6     5     2
```

or

$h(x) = x^5 + 2x^4 - 4x^3 - 6x^2 + 5x + 1$

deconv(*p*, *q*)

Let $p(x)$ and $q(x)$ be two polynomial of degree n and m respectively, to find $h(x) = p(x)/q(x)$ using **[*h*, *r*]=deconv(*p*, *q*)**
Returns two vectors. The first vector contains the coefficients of the quotient and the second vector contains the coefficients of the remainder polynomial.

Example 2.3: - consider the polynomials
$p(x) = x^4 - 4x^2 + 2x + 1$ and $q(x) = x^2 + 2$
then to find $h(x) = p(x)/q(x)$ write Matlab code as follows

```
p=[1 0 -4 2 1];
q=[1 0 2];
[h,r]=deconv(p,q)
```

result is

```
h =
     1     0    -6
r =
     0     0     0     2    13
```

or

$h(x) = x^2 - 6$ and $r(x) = 2x + 13$

2.3 Polynomial Algebraic

roots(*p*)

Let $p(x)$ is a polynomial the MATLAB function for determining the roots of a polynomial is the **roots** function. **r=roots(*p*)** determines the roots of the polynomial represented by the coefficient vector *p*. The roots function returns a column vector containing the roots of the polynomial, the number of roots is equal to the degree of the polynomial.

Example 2.4: - consider the polynomial $p(x) = x^4 - 4x^2 + 2x + 1$ then to find the roots of *p* write Matlab code as follows

```
p=[1 0 -4 2 1];
r=roots(p)
```

result is

```
r =
   -2.1701
    1.4812
    1.0000
   -0.3111
```

are four real roots of polynomial *p*.

poly(*r*)

When the roots of a polynomial are known, the coefficients of the polynomial are determined when all the linear terms are multiplied, we can use the poly function *p*=**poly(***r***)** Determines the coefficients of the polynomial whose roots are contained in the vector *r*. The output of the function is a row vector containing the polynomial coefficients.

Example 2.5: - consider the polynomial $r = [1\ 2\ \ 0\ -4]$,then to find the $p(x)$ with roots *r*, write Matlab code as follows

```
r=[1 2 0 -4];
p=poly(r)
```

result is

```
p =
     1     1   -10     8     0
```

or

$p(x) = x^4 + x^3 - 10x^2 + 8x$

polyval (*p*, *x*)

The value of a polynomial can be computed using the *polyval* function
y=**polyval(***p***,** *x***)** it evaluates a polynomial with coefficients *p* for the values in *x*. The result is a matrix the same size of *x*.

Example 2.6: - consider the polynomial $p(x) = x^4 - 4x^2 + 2x + 1$ and *x*=2, then to find the $p(2)$ write Matlab code as follows

```
p=[1 0 -4 2 1];
x=2;
y=polyval(p,x)
```

result is

```
y =
      5
```

2.4 Polynomial Calculus

polyder(p)

Let $p(x)$ is a polynomial the MATLAB function for determining the derivative of $p(x)$ is *q***=polyder(***p***)** returns the derivative of the polynomial represented by the coefficients in *p*,

$$q(x) = \frac{d}{dx}p(x).$$

h = **polyder(***p***,***q***)** returns the derivative of the product of the polynomials *p* and *q*,

$$h(x) = \frac{d}{dx}[p(x) * q(x)].$$

[*h***,***r***] = polyder(***p***,***q***)** returns the derivative of the quotient of the polynomials *p* and *q*,

$$\frac{h(x)}{r(x)} = \frac{d}{dx}\frac{p(x)}{q(x)}$$

Example 2.7: - consider the polynomial $p(x) = x^4 - 4x^2 + 2x + 1$, then to find the derivative of *p* write Matlab code as follows

```
p=[1 0 -4 2 1];
q=polyder(p)
```

result is

```
q =
      4      0     -8      2
```

or
$q(x) = 4x^3 - 8x + 2$

polyint

Let $p(x)$ is a polynomial the MATLAB function for determining the integration of $p(x)$ is *q* = **polyint**(*p*,*k*) returns the integral of the polynomial represented by the coefficients in *p* using a constant of integration *k*. *q* = **polyint**(*p*) assumes a constant of integration *k* = 0.

Example 2.8: - consider the polynomial $p(x) = 4x^3 - 3x^2 + 8x + 1$, then to find the integration of *p* write Matlab code as follows

```
p=[4 -3 8 1];
q=polyint(p)
```

result is

```
q =
     1    -1      4      1      0
```

or
$q(x) = x^4 - x^3 - 4x^2 + x$

polyfit

polyfit returns a vector of coefficients representing the fitted polynomial in descending order.

p = polyfit(x,y,n)

where x and y are vectors of the same length representing the x- and y-coordinates of the data points, respectively, n is the degree of the polynomial, and p is a vector of coefficients representing the fitted polynomial in descending order.

[p,S] = polyfit(x,y,n)

also returns a structure S that can be used as an input to polyval to obtain error estimates.

[p,S,mu] = polyfit(x,y,n)

performs centering and scaling to improve the numerical properties of both the polynomial and the fitting algorithm. This syntax additionally returns mu, which is a two-element vector with centering and scaling values. mu(1) is mean(x), and mu(2) is std(x). Using these values, polyfit centers x at zero and scales it to have unit standard deviation,

Example 2.9:-

Let $x = [1, 2, 3, 4, 5]$ and $y = [2,5,10,17,26]$ then

p = polyfit(x,y,2)

Symbolic Processing

We have focused on the use of Matlab to perform numerical operations, involving numerical data represented by double precision floating point numbers. We also given some consideration to the manipulation of text, represented by character strings. In this section, we introduce the use of Matlab to perform *symbolic processing* to manipulate mathematical expressions, in much the way that we do with pencil and paper.
The objective of symbolic processing is to obtain what are known as *closed form* solutions, expressions that don't need to be iterated or updated in order to obtain answers. An understanding of these solutions often provides better physical and mathematical insight into the problem under investigation.

2.1 Declaring Symbolic Variables and Constants

To enable symbolic processing, the variables and constants involved must first be declared as *symbolic objects*.

`Syms` *`var1`*`,`*`var2`*`,……`

For example, to create the symbolic variables with names x and y:

```
>> syms x y
```

If x and y are to be assumed to be real variables, they are created with the command:

```
>> syms x y real
```

`Sym(var)`

To declare symbolic constants, the sym function is used. Its argument is a string containing the name of a special variable, a numeric expression, or a function evaluation. It is used in an assignment statement which serves as a declaration of a symbolic variable for the assigned variable. Examples include:

```
>> pi = sym('pi');

>> delta = sym('1/10');

>> sqroot2 = sym('sqrt(2)');
```

If the symbolic constant pi is created this way, it replaces the special variable **pi** in the workspace. The advantage of using symbolic constants is that they maintain full accuracy until a numeric evaluation is required.

Example 2.1: -

```
 >> x=sym(1/2)          >> x*y            >> x/y
 x = 1/2                ans = 1/3         ans = 3/4
 >> y=sym(2/3)          >> x+y            >> x^2
 y = 2/3                ans = 7/6         ans = 1/4
```

Symbolic variables and constants are represented by the data type symbolic object.

```
>> whos
Name        Size        Bytes        Class
delta       1x1          132       sym object
pi          1x1          128       sym object
sqroot2     1x1          138       sym object
x           1x1          126       sym object
```

2.2 Symbolic Expressions

Symbolic variables can be used in expressions and as arguments of functions in much the same way as numeric variables have been used. The operators `+ - * / ^` and the built-in functions can also be used in the same way as they have been used in numeric calculations.

Example 2.2:-

```
>> syms x y
f = x^2*y + 5*x*sqrt(y)
f =
 x^2*y + 5*x*y^(1/2)
```

```
syms s t A
g = s^2 + 4*s + A
g =
 s^2 + 4*s + A

 >> h=f*g
 h =
 (x^2*y + 5*x*y^(1/2))*(s^2 + 4*s + A)
```

The variable *x* is the default independent variable, but as can be seen with the expressions above, other variables can be specified to be the independent variable. It is important to know which variable is the independent variable in an expression. The command to find the independent variable is:

symvar (S)

Finds the symbolic variables in a symbolic expression or matrix S by returning a string containing all of the symbolic variables appearing in S. The variables are returned in alphabetical order and are separated by commas. If no symbolic variables are found, **symvar** returns the empty string.
Use example 2.2

```
>> symvar(f)
ans =
     [x,y]
>> symvar(z)
ans =
     [A,s,t]
```

2.3 Manipulating Polynomial Expressions

In the examples above, symbolic variables were declared and were used in symbolic expressions to create polynomials. We now wish to manipulate these polynomial expressions algebraically.
The Matlab commands for this purpose include:

expand(S)

Expands each element of a symbolic expression S as a product of its factors. **expand** is most often used on polynomials, but also expands trigonometric, exponential and logarithmic functions.

Example 2.3:-

```
syms x;
expand((x-2)*(x-4))
ans = x^2 - 6*x + 8

syms x y;
expand(cos(x+y))
ans = cos(x)*cos(y) - sin(x)*sin(y)

syms a b;
expand(exp((a+b)^2))
ans = exp(2*a*b)*exp(a^2)*exp(b^2)

syms t;
expand([sin(2*t), cos(2*t)])
ans =[ 2*cos(t)*sin(t), cos(t)^2 - sin(t)^2]
```

factor(S)

Factors each element of the symbolic matrix S.

Example 2.4:-

```
syms x y;
factor(x^3-y^3)
ans = (x - y)*(x^2 + x*y + y^2)

syms a b;
factor([a^2 - b^2, a^3 + b^3])
ans = [ (a - b)*(a + b), (a + b)*(a^2 - a*b + b^2)]

x=sym(5678)
x = 5678
factor(x)
ans = 2*17*167
```

simplify(S)

Simplifies each element of the symbolic matrix S.

Example 2.5:-

```
syms s
H = (s^3 +2*s^2 +5*s +10)/(s^2 + 5);
H = simplify(H)
H = s+2
```

```
syms x;
simplify(sin(x)^2 + cos(x)^2)
ans =1
```

```
syms a b c;
simplify(exp(c*log(sqrt(a+b))))
ans =(a + b)^(c/2)
```

```
S = [(x^2 + 5*x + 6)/(x + 2), sqrt(16)];
R = simplify(S)
R =[ x + 3, 4]
```

[n , d] = numden(S)

Returns two symbolic expressions that represent the numerator expression num and the denominator expression den for the rational representation of the symbolic expression S.

Example 2.6:-

```
[n, d] = numden(sym(4/5))
ans
n =4
d =5
```

```
syms x y;
[n,d] = numden(x/y + y/x)
ans
n = x^2 + y^2
d = x*y
```

```
 syms s
G = s+4 + 2/(s+4) + 3/(s+2);
[N, D] = numden(G)
```

```
N = s^3+10*s^2+37*s+48
D =(s+4)*(s+2)
```

collect(f)

views f as a polynomial in its symbolic variable, say x, and collects all the coefficients with the same power of x. A second argument can specify the variable in which to collect terms if there is more than one candidate.

Example 2.7:-

```
f=(x-1)*(x-2)*(x-3)
collect(f)
ans= x^3-6*x^2+11*x-6

f=x*(x*(x-6)+11)-6
collect(f)
ans= x^3-6*x^2+11*x-6

f=(1+x)*t + x*t
collect(f)
ans=2*x*t+t
```

subs(S,old,new)

Symbolic substitution, replacing symbolic variable old with symbolic variable new in the symbolic expression S.

Example 2.8:-

```
f = 2*x^2 - 3*x + 1
subs(f,2)
ans =3

syms x y
f = x^2*y + 5*x*sqrt(y)
subs(f, x, 3)
ans = 9*y+15*y^(1/2)
```

```
syms s
H = (s+3)/(s^2 +6*s + 8);
G = subs(H,s,s+2)
G = (s+5)/((s+2)^2+6*s+20)

E = s^3 -14*s^2 +65*s -100;
F = subs(E,s,7.1)
F =13671/1000
```

2.5 Polynomial and Solving Equations

Polynomials divisor function
[q, r] = quorem(p, h) Divided p by h and return quotient q and remainder r.

Example 2.14: -

```
syms x
p=x^3-4*x^2+3*x+1;
h=x^2+1;
[q, r] = quorem(p, h)
q = x - 4
r =2*x + 5
```

Coefficients of polynomial
C = **coeffs(p)** returns the coefficients of the polynomial p with respect to all the in determinates of p.
C = **coeffs(p, x)** returns the coefficients of the polynomial p with respect to x.

Example 2.15: -

```
syms x
 f=2*x^3-6*x+2;
coeffs(f)
ans =[ 2, -6, 2]
```

Example 2.16: -

```
syms x y
z = 3*x^2*y^2 + 5*x*y^3;
coeffs(z)
coeffs(z,x)
```

```
ans =[ 5, 3]
ans =[ 5*y^3, 3*y^2]
```

Conversions

sym2poly(P):- Converts from a symbolic polynomial P to a row vector containing the polynomial coefficients.

poly2sym(p) :-Converts from a polynomial coefficient vector p to a symbolic polynomial in the variable x. poly2sym(p,v) uses the symbolic the variable v.

Example 2.17: - consider the polynomial $A(s) = s^3 + 4s^2 - 7s - 10$
In Matlab:

```
a = [1 4 -7 -10];
A = poly2sym(a,s)
A = s^3+4*s^2-7*s-10
```

Example 2.18: - For the polynomial $B(s) = 4s^3 - 2s^2 + 5s - 16$

```
syms s
B = 4*s^3 -2*s^2 +5*s -16;
b = sym2poly(B)
b = 4 -2 5 -16
```

double

The function double(c) converts the symbolic object c (constants, scalar, or matrix) into a double precision floating point variable.

Example 2.19: -

```
 a=sym(2/3)
 b=sym(1/5);
 a+b=13/15
 double(a+b)= 0.8667
```

Solving Equations

You can solve equations involving variables with **solve**.

Example 2.20: - to find the solutions of the quadratic equation $x^2 - 2x - 4 = 0$, type

```
solve('x^2 - 2*x - 4 = 0')
ans =
[ 5^(1/2)+1]
[ 1-5^(1/2)]
```

Or

```
syms x
f=x^2-2*x-4;
solve(f)
ans =
 1 - 5^(1/2)
 5^(1/2) + 1
```

Note that the equation to be solved is specified as a string; that is, it is surrounded by single quotes. The answer consists of the exact (symbolic) solutions $1 \pm\sqrt{5}$. To get numerical solutions, type **double(ans)**, or **vpa(ans)** to display more digits.

```
double(solve(f))
ans =
   -1.2361
    3.2361
```

```
Or
```

```
vpa(solve(f))
ans =
 -1.2360679774997896964091736687313
  3.2360679774997896964091736687313
```

The command **solve** can solve higher-degree polynomial equations, as well as many other types of equations. It can also solve equations involving more than one variable. If there are fewer equations than variables, you should specify (as strings) which variable(s) to solve for.

Example 2.21: - to solve $2x - \log y = 1$ for y in terms of x.

```
syms x y
solve(2*x - log(y) ==1, x)
ans =
log(y)/2 + 1/2
```

You can specify more than one equation as well.

Example 2.22: -

```
syms x y
[xsol, ysol] = solve(x^2 - y == 2, y - 2*x ==5,[x,y])
```

Calculus Applications

The Symbolic Math Toolbox provides functions to do the basic operations of calculus; differentiation, limits, integration, summation, and Taylor series expansion.

4.1 Limits

limit:-Compute limit of symbolic expression

Syntax

```
limit(expr, x, a)
limit(expr, a)
limit(expr)
limit(expr, x, a, 'left')
limit(expr, x, a, 'right')
```

Description

`limit(expr,x,a)`:- computes bidirectional limit of the symbolic expression `expr` when `x` approaches *a*.

`limit(expr,a)`:- computes bidirectional limit of the symbolic expression `expr` when the default variable approaches *a*.

`limit(expr)`:- computes bidirectional limit of the symbolic expression `expr` when the default variable approaches `0`.

`limit(expr,x,a,'left')`:-computes the limit of the symbolic expression `expr` when `x` approaches *a* from the left.

`limit(expr,x,a,'right')`:- computes the limit of the symbolic expression `expr` when `x` approaches *a* from the right.

Example 4.1:- Find the following limits

$$1)\ \lim_{x\to 2}\frac{x^2-4}{x-2} \qquad 2)\ \lim_{y\to 0}(xy+x) \qquad 3)\ \lim_{x\to 0^+}\frac{1}{x} \qquad 4)\ \lim_{t\to\infty}\frac{1-t^2}{3t^2+t} \qquad 5)\ \lim_{x\to 0^-}\frac{|x|}{x}$$

1)
```
syms x y
 limit((x^2-4)/(x-2),2)
 ans =4
or
limit((x^2-4)/(x-2),x,2)
 ans =4
```

2)
```
limit(x*y+x,y,0)
  ans = x

if we write
  limit(x*y+x,0)
  ans = 0
```

3)
```
limit(1/x,x,0,'right')
  ans = Inf
```

4)
```
syms t
 limit((1-t^2)/(3*t^2+t),t,inf)
 ans =-1/3
```

5)
```
limit(abs(x)/x,x,0,'left')
  ans = -1
```

4.2 Differentiation

diff:- Differentiate symbolic expression

Syntax
```
diff(expr)
diff(expr, v)
diff(expr, n)
diff(expr, v, n)
```

Description

`diff(expr)`:- differentiates a symbolic expression `expr` with respect to its free variable as determined by `symvar`.

`diff(expr, v)`:- differentiate `expr` with respect to `v`.

`diff(expr, n)`:- differentiates `expr` *n* times. *n* is a positive integer.

`diff(expr, v, n)`:- differentiate `expr` with respect to `v` `n` times.

Example 4.2: - Find the following derivative

$$1)\ \frac{d}{dx}(e^x \sin ax) \quad 2)\ \frac{d^3}{dt^3}(t^3 + \tan\ t) \quad 3)\frac{\partial}{\partial y}(x^2 + y^2 - 3xy)$$

```
 syms x y t a
1) diff(exp(x)*sin(a*x))
   ans = exp(x)*sin(a*x) + a*exp(x)*cos(a*x)

  diff(exp(x)*sin(a*x),x)
  ans = exp(x)*sin(a*x) + a*exp(x)*cos(a*x)
```

```
2) diff(t^3+tan(t),t,3)
   ans =2*(tan(t)^2 + 1)^2 + 4*tan(t)^2*(tan(t)^2 + 1) + 6

3) diff(x^2+y^2-3*x*y,y)
   ans =2*y - 3*x
```

jacobian

Compute Jacobian matrix

Syntax

`jacobian(f, v)`

Description

`jacobian(f, v)`:- computes the Jacobian of the scalar or vector `f` with respect to `v`. The `(i, j)`-th entry of the result is $\partial f(i)/\partial v(j)$. If `f` is scalar, the Jacobian of `f` is the gradient of `f`. If `v` is a scalar, the result equals to `diff(f, v)`.

Example 4.3: - Let $x = rcos\theta$ and $y = rsin\theta$ then find $J = \frac{\partial(x,y)}{\partial(r,\theta)}$.

```
 syms x y r th
  J=jacobian([x; y], [r th])
  x=r*cos(th);
  y=r*sin(th);
  J=jacobian([x; y], [r th])
  J = [ cos(th), -r*sin(th)]
      [ sin(th),  r*cos(th)]
  det(J)
  ans = r*cos(th)^2 + r*sin(th)^2
  simplify(ans)
  ans = r
```

4.3 Integration

int:-Integrate symbolic expression

Syntax

```
int(expr)
int(expr, v)
int(expr, a, b)
int(expr, v, a, b)
```

Description

`int(expr)` :- returns the indefinite integral of `expr` with respect to its symbolic variable as defined by `symvar`.

`int(expr,v)` :- returns the indefinite integral of `expr` with respect to the symbolic scalar variable `v`.

`int(expr,a,b)` :- returns the definite integral from *a* to *b* of `expr` with respect to the default symbolic variable. *a* and *b* are symbolic or double scalars.

`int(expr,v,a,b)` :- returns the definite integral of `expr` with respect to `v` from *a* to *b*.

Example 4.4: - Find the following integral

$$1)\ \int \frac{1}{2+x^2}dx \qquad 2)\ \int xe^x dx \qquad 3)\ \int_0^1 \sqrt{1-x^2}dx$$

```
1) syms x
   int(1/(2+x^2),x)
   ans =(2^(1/2)*atan((2^(1/2)*x)/2))/2

2) int(x*exp(x))
   ans = exp(x)*(x - 1)

3) int(sqrt(1-x^2),x,0,1)
   ans = pi/4
```

4.4 Symbolic Summation

symsum:- Evaluate symbolic sum of series

Syntax

```
r = symsum(expr)
r = symsum(expr, v)
r = symsum(expr, a, b)
r = symsum(expr, v, a, b)
```

Description

`r = symsum(expr)` :- evaluates the sum of the symbolic expression `expr` with respect to the default symbolic variable `defaultVar` determined by `symvar`. The value of the default variable changes from `0` to `defaultVar - 1`.

`r = symsum(expr,v)`:- evaluates the sum of the symbolic expression `expr` with respect to the symbolic variable `v`. The value of the variable `v` changes from `0` to `v - 1`.
`r = symsum(expr,a,b)`:- evaluates the sum of the symbolic expression `expr` with respect to the default symbolic variable `defaultVar` determined by `symvar`. The value of the default variable changes from `a` to `b`.
`r = symsum(expr,v,a,b)`:- evaluates the sum of the symbolic expression `expr` with respect to the symbolic variable `v`. The value of the default variable changes from `a` to `b`.

Example 4.5:- Find the sum of the following series

$$1)\sum_{i=0}^{n} i \qquad 2)\sum_{n=1}^{\infty}\frac{1}{n^2} \qquad 3)\sum_{k=0}^{n} x^k \qquad 4)\sum_{k=1}^{n}\frac{1}{k}-\frac{1}{k+1}$$

```
1) syms x n k
    symsum(k,1,n)
    ans = (n*(n + 1))/2
 or
   symsum(n+1)
   ans = n^2/2 + n/2

2) s1=symsum(1/k^2,1,inf)
   s1 = pi^2/6

3) s2=symsum(x^k,k,0,inf)
   s2 = piecewise([1 <= x, Inf], [abs(x) < 1, -1/(x - 1)])

4) symsum(1/k-1/(k+1),k,1,n)
  ans = psi(n + 1) - psi(n + 2) + 1
  simplify(ans)
  ans = 1 - 1/(n + 1)
```

4.5 Taylor Series

taylor:- Taylor series expansion

Syntax

```
taylor(f)
taylor(f, n)
taylor(f, a)
taylor(f, n, v)
taylor(f, n, v, a)
```

Description

`taylor(f)` :- returns the fifth order Maclaurin polynomial approximation to `f`.

`taylor(f,n)` :– returns the (`n−1`)-order Maclaurin polynomial approximation to `f`. Here `n` is a positive integer.

`taylor(f,a)` :- returns the fifth order Taylor series approximation to `f` about point `a`. Here `a` is a real number. If `a` is a positive integer or if you want to change the expansion order, use `taylor(f,n,a)` to specify the base point and the expansion order.

`taylor(f,n,v)` :– returns the (`n−1`)-order Maclaurin polynomial approximation to `f`, where `f` is a symbolic expression representing a function and `v` specifies the independent variable in the expression. `v` can be a string or symbolic variable.

`taylor(f,n,v,a)` :– returns the Taylor series approximation to `f` about `a`. The argument `a` can be a numeric value, a symbol, or a string representing a numeric value or an unknown. If `a` is a symbol or a string, do not omit `v`.

Example 4.6:- Find Maclaurin series of e^x and Taylor series of it about $x_0 = 1$.

```
syms x
taylor(exp(x))
ans = x^5/120 + x^4/24 + x^3/6 + x^2/2 + x + 1

taylor(exp(x),10)
ans = x^9/362880 + x^8/40320 + x^7/5040 + x^6/720 + x^5/120
+ x^4/24 + x^3/6 + x^2/2 + x + 1

taylor(exp(x),3,1)
ans = exp(1) + exp(1)*(x - 1) + (exp(1)*(x - 1)^2)/2
```

4.8 Functional composition

compose:-Functional composition

Syntax

```
compose(f,g)
compose(f,g,z)
compose(f,g,x,z)
compose(f,g,x,y,z)
```

Description

`compose(f,g)` :- returns `f(g(y))` where `f = f(x)` and `g = g(y)`. Here `x` is the symbolic variable of `f` as defined by `symvar` and `y` is the symbolic variable of `g` as defined by `symvar`.
`compose(f,g,z)` :- returns `f(g(z))` where `f = f(x)`, `g = g(y)`, and `x` and `y` are the symbolic variables of `f` and `g` as defined by `symvar`.
`compose(f,g,x,z)` :- returns `f(g(z))` and makes `x` the independent variable for `f`. That is, if `f = cos(x/t)`, then `compose(f,g,x,z)` returns `cos(g(z)/t)` whereas `compose(f,g,t,z)` returns `cos(x/g(z))`.
`compose(f,g,x,y,z)` :- returns `f(g(z))` and makes `x` the independent variable for `f` and `y` the independent variable for `g`. For `f = cos(x/t)` and `g = sin(y/u)`, `compose(f,g,x,y,z)` returns `cos(sin(z/u)/t)` whereas `compose(f,g,x,u,z)` returns `cos(sin(y/z)/t)`.

Examples 4.9: -

Suppose

```
syms x y z t u;
f = 1/(1 + x^2); g = sin(y); h = x^t; p = exp(-y/u);
```

Then

```
a = compose(f,g)
b = compose(f,g,t)
c = compose(h,g,x,z)
d = compose(h,g,t,z)
e = compose(h,p,x,y,z)
f = compose(h,p,t,u,z)
```

returns:

```
a = 1/(sin(y)^2 + 1)
b = 1/(sin(t)^2 + 1)
c = sin(z)^t
d = x^sin(z)
e =(1/exp(z/u))^t
f = x^(1/exp(y/z))
```

Linear Algebra

5.1 Solving Linear Systems

Using mldivide or x=A\b

Suppose that A is a non-singular n × n matrix and b is a column vector of length n. Then typing `x = A\b` numerically computes the unique solution to `A*x = b`. Type help mldivide for more information.

Example 5.1:- use MATLAB to solve the following linear system

$$\begin{aligned} x-2y+z &= -1\\ x+y-z &= 0\\ 2x+y-3z &= 2 \end{aligned}$$

solve:-

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

X =
   -0.8000
   -0.6000
   -1.4000
```

Using inv(A)

Suppose that A is a non-singular n × n matrix and b is a column vector of length n. Then can solved linear system $Ax=b$ by $x=inv(A)*b$

Example 5.1 can be solving by

```
X=inv(A)*b
```

Using linsolve

X = linsolve(A,B) solves the linear system AX = B using LU factorization with partial pivoting when A is square and QR factorization with column pivoting otherwise. The number of rows of A must equal the number of rows of B.

5.2 Calculating Eigenvalues and Eigenvectors

The eigenvalues of a square matrix A are calculated with `eig(A)`. The command `[U, R] = eig(A)` calculates both the eigenvalues and eigenvectors. The eigenvalues are the diagonal elements of the diagonal matrix R, and the columns of U are the eigenvectors.
Here is an example illustrating the use of eig.

Example 5.2:-

```
>> A = [3 -2 0; 2 -2 0; 0 1 1];
>> eig(A)
ans =
      1
     -1
      2
>> [U, R] = eig(A)
U =
        0   -0.4082   -0.8165
        0   -0.8165   -0.4082
   1.0000    0.4082   -0.4082
R =
   1    0    0
   0   -1    0
   0    0    2
```

The eigenvector in the first column of U corresponds to the eigenvalue in the first column of R, and so on.

5.3 MATLAB Linear Algebra Functions

Euclidean Norm

The length of a vector is called the norm of the vector. From Euclidean geometry, the distance between two points is the square root of the sum of the squares of the distances in each dimension. Thus, the notation and definition of the Euclidean norm is

$$\|X\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

Note that the norm can be defined in terms of the inner product:

$$\|X\| = \sqrt{(X, X)}$$

Vector norm
`n = norm(X)` returns the Euclidean Norm of vector X.
`n = norm(X,p)` returns a different kind of norm, depending on the value of p.

Example 5.3: - Let `a=(2 3 -4)` and `b=(7 -4 5)` then find the norm and unit vector for each of a and b then find the angle between a and b.

```
>> a=[2 3 -4];

>> b=[7 -4 5];

>> na=norm(a)

na =   5.3852

>> nb=norm(b)

nb =   9.4868

>> ua=a/norm(a)

ua =   0.3714    0.5571   -0.7428

>> ub=b/norm(b)

ub =   0.7379   -0.4216    0.5270

>> theta = acos(dot(a,b)/(norm(a)*norm(b)))

theta =  1.9309
```

Rank of Matrix

rank(A) :- Returns the rank of the matrix A, which is the number of independent rows of A.

Example 5.4:- Let `M = [0 2 2 3 -4; -2 4 2 -1 -6; 3 -4 -1 2 8];` then

```
rank(M)=3
```

Reduced row echelon form

R = rref(A) produces the reduced row echelon form of A using Gauss Jordan elimination with partial pivoting.

Example 5.5:- Use *rref* to solve the following linear system

$$\begin{aligned} x - 2y + z &= -1 \\ x + y - z &= 0 \\ 2x + y - 3z &= 2 \end{aligned}$$

sol:-
```
>> A=[1 -2 1;1 1 -1;2 1 -3];
>> b=[-1;0;2];
>> R=rref([A b]);
>> Sol=R(:,4)'
Sol = -0.8000   -0.6000   -1.4000
```

Other functions

There are useful function listed in below table

1	C = dot(A,B)	returns the scalar dot product of A and B.
2	C = cross(A,B)	returns the cross product of A and B.
3	L = tril(A)	returns the lower triangular part of A.
4	U = triu(A)	returns the Upper triangular part of A.
5	b = trace(A)	is the sum of the diagonal elements of the matrix A.
6	Q = orth(A)	returns an orthonormal basis for the range of A. The columns of Q are vectors, which span the range of A. The number of columns in Q is equal to the rank of A.

Solving Differential Equation

6.1 Single Differential Equation

The function dsolve computes symbolic solutions to ordinary differential equations. The equations are specified by symbolic expressions containing the letter D to denote differentiation. The symbols D2, D3, ... DN, correspond to the second, third, ..., Nth derivative, respectively. Thus, D2y is the Symbolic Math Toolbox equivalent of d^2y/dt^2. The dependent variables are those preceded by D and the default independent variable is t. Note that names of symbolic variables should not contain D. The independent variable can be changed from t to some other symbolic variable by including that variable as the last input argument.

Syntax

```
dsolve('eq','cond1','cond2',...,'v')

or
dsolve(eq,cond1,'cond2)
```

Example 6.1:- Find the general solution of the first order differential equation

$$\frac{dy}{dt} + y = te^t$$

sol:-

```
>> dsolve('Dy + y = t*exp(t)')

ans =
     1/2*t*exp(t)-1/4*exp(t)+exp(-t)*C1

Or

>> syms y(t)
>> dsolve(diff(y)+y==t*exp(t))
ans =
(exp(t)*(2*t - 1))/4 + C1*exp(-t)
```

Example 6.2:- Find the partical solution of the first order differential equation

$$\frac{dy}{dt} = 1 + y^2, \quad y(0) = 1$$

```
dsolve('Dy=1+y^2')
```

uses *y* as the dependent variable and *t* as the default independent variable.

The output of this command is

```
ans = tan(t+C1)
```

To specify an initial condition, use

```
y = dsolve('Dy=1+y^2','y(0)=1')
```

This produces

```
y =tan(t+1/4*pi)


or
>> dsolve(diff(y)==1+y^2,y(0)==1)
ans =
tan(t + pi/4)
```

Example 6.3: - Nonlinear equations may have multiple solutions, even when initial conditions are given:

```
x = dsolve('(Dx)^2+x^2=1','x(0)=0')
```

results in

```
x =
   [ sin(t)]
   [ -sin(t)]
Or
>> syms x(t)
>> x = dsolve(diff(x)^2+x^2==1,x(0)==0)
x =
 cosh(t*1i + (pi*1i)/2)
 cosh(t*1i - (pi*1i)/2)
```

Example 6.4:- Here is a second order differential equation with two initial conditions.

The commands

```
y = dsolve('D2y=cos(2*x)-y','y(0)=1','Dy(0)=0', 'x');
simplify(y)
```

produce

```
ans = 4/3*cos(x)-2/3*cos(x)^2+1/3
or
>> y = dsolve(diff(y,2)==cos(2*x)-y,y(0)==1',diff(y(0))==0)
y =
(5*cos(x))/3 + C20*sin(x) + sin(x)*(sin(3*x)/6 + sin(x)/2)
- (2*cos(x)*(6*tan(x/2)^2 - 3*tan(x/2)^4 +
1))/(3*(tan(x/2)^2 + 1)^3)
>> simplify(y)
ans =
(4*cos(x))/3 - (2*cos(x)^2)/3 + C20*sin(x) + 1/3
```

6.2 Several Differential Equations

The function dsolve can also handle several ordinary differential equations in several variables, with or without initial conditions. For example, here is a pair of linear, first-order equations.

```
dsolve('eq1','eq2','eq3', ...,'cond1','cond2',...,'v')

or
dsolve(eq1,eq2,eq3,...,cond1,'cond2,...,)
```

Example 6.5: - solve linear system of differential equations

$$x' = 3x + 4y$$
$$y' = -4x + 3y$$

```
S = dsolve('Dx = 3*x+4*y', 'Dy = -4*x+3*y')
```

The computed solutions are returned in the structure S. You can determine the values of f and g by typing

```
x = S.x
x = exp(3*t)*(C1*sin(4*t)+C2*cos(4*t))
y = S.y
y = exp(3*t)*(C1*cos(4*t)-C2*sin(4*t))
```

If you prefer to recover f and g directly as well as include initial conditions, type

```
[x,y] = dsolve('Dx=3*x+4*y,Dg =-4*x+3*y', 'x(0) = 0,y(0) =
1')
f = exp(3*t)*sin(4*t)
g = exp(3*t)*cos(4*t)


or
>> syms x(t) y(t)
>> S = dsolve(diff(x) == 3*x+4*y, diff(y) == -4*x+3*y);
>> S.x
ans =
C22*cos(4*t)*exp(3*t) + C21*sin(4*t)*exp(3*t)
>> S.y
ans =
C21*cos(4*t)*exp(3*t) - C22*sin(4*t)*exp(3*t)
```