Name - Saiyed Mohd Ahmad
Enrolment No. - GK1416
Faculty No. - 20COB271
Serial No. - 2

ANSWER NO -1

Vpa

- The vpa (a) is used to compute the elements of the input ‘a’ till ‘n’ significant digits.
- By default, the value of significant digits is 5 for the vpa function.

Example:-

```
 ps =
 (8242085728639383*x^4)/2251799813685248 -
(1030260716079923*x^3)/140737488355328 +
(5181631624232949*x^2)/9007199254740992 +
(6946677822581147*x)/2251799813685248 +
5667700847528145/40564819207303340847894502572032

>> vpa(ps,5)

ans =

3.6602*x^4 - 7.3204*x^3 + 0.57528*x^2 + 3.0849*x + 1.3972e-16
```

Poly2sym

poly2sym Polynomial coefficient vector to symbolic polynomial.
poly2sym(C) is a symbolic polynomial in 'x' with coefficients from the vector C.
poly2sym(C, V) uses the symbolic variable specified by the second argument.

Example:

```
 poly2sym([1 0 -2 -5]) is  x^3-2*x-5

t = sym('t')
 poly2sym([1 0 -2 -5],t) returns
t^3-2*t-5
```

diff(f,n)

Df = diff (f,n) **computes the n th derivative of f** with respect to the symbolic scalar variable determined by symvar. example Df = diff (f,var) differentiates f with respect to the differentiation parameter var. var can be a symbolic scalar variable, such as x, a symbolic function, such as f (x), or a derivative function, such as diff (f (t),t).

Example:-

```
>> syms x;
>> f=sin(exp(x));
>> diff(f,4)

ans =

cos(exp(x))*exp(x) - 6*cos(exp(x))*exp(3*x) - 7*sin(exp(x))*exp(2*x) +
sin(exp(x))*exp(4*x)
```

pretty

Pretty print a symbolic expression.
pretty is not recommended. Use live scripts instead.
Live scripts provide full math rendering while pretty uses plain-text formatting.
pretty(S) prints the symbolic expression S in a format that
resembles type-set mathematics.

Example:-

```
>> syms a x;
>> diff(a*atan(x),4)

ans =

(24*a*x)/(x^2 + 1)^3 - (48*a*x^3)/(x^2 + 1)^4

>> pretty(ans)
                  3
  24 a x     48 a x
--------- - ---------
  2    3     2     4
```

(x + 1)   (x + 1)

Int :-

int(S,a,b) is the definite integral of S with respect to its symbolic variable from a to b. a and b are each double or symbolic scalars. The integration interval can also be specified using a row or a column vector with two elements, i.e., valid calls are also int(S,[a,b]) or int(S,[a b]) and int(S,[a;b]).

Example:-

```
>> syms x;
>> f=sin(x)+cos(x);
>> a=int(f,0,pi)

a =

2
```

Solve:-

The **solve function is used for solving algebraic equations**. In its simplest form, the solve function takes the equation enclosed in quotes as an argument. If the equation involves multiple symbols, then MATLAB by default assumes that you are solving for x, however, the solve function has another form − where, you can also mention the variable.

Example:-

```
syms x y
 [Sx,Sy] = solve(x^2 + x*y + y == 3,x^2 - 4*x + 3 == 0)
Returns
Sx =
 1
 3

Sy =
   1
 -3/2
```

Dsolve:-

dsolve('eq1','eq2',...,'cond1','cond2',...,'v') symbolically solves the ordinary differential equations eq1, eq2,... using v as the independent variable. Here cond1,cond2,... specify boundary or initial conditions or both. You also can use the following syntax: dsolve('eq1, eq2',...,'cond1,cond2',...,'v').

Example:-

Solve
dy /dt =2*y+t ,y(0)=1;
Solution:

```
>> dsolve('Dy=2*y+t','y(0)=1','t')
ans = (5*exp(2*t))/4 - t/2 - ¼
```

<u>ANSWER NO 2</u>

Laplace:-

Laplace function is used in MATLAB to calculate the **laplace transform of a function**. We can calculate the Laplace transform w.r.t to the default transformation variable‘s’or the variable we define as the transformation variable.

Example:-

```
>> syms t;
>> f=sin(t)+exp(t)
 f =
exp(t) + sin(t)
 >> a=laplace(f)
 a =
 1/(s - 1) + 1/(s^2 + 1)
```

Inverse Laplace transform(Ilaplace):-

F = ilaplace(L) is the inverse Laplace transform of the sym L with default independent variable s. The default return is a function of t. If L = L(t), then ilaplace returns a function of x: F = F(x).

Example:-

```
>> syms s;
>> f=1/(s+3)+5/(s-9)-25/(s^2+25)
 f =
 1/(s + 3) + 5/(s - 9) - 25/(s^2 + 25)
>> a=ilaplace(f)
 a =
 exp(-3*t) + 5*exp(9*t) - 5*sin(5*t)
```

Residue:-

The Matlab Residue Command ,**allows one to do partial fraction expansion**. > help residue RESIDUE Partial-fraction expansion (residues). [R,P,K] = RESIDUE(B,A) finds the residues, poles and direct term of ,

If there are no multiple roots,

```
       B(s)       R(1)       R(2)             R(n)
       ----  =  -------- + -------- + ... + -------- + K(s)
       A(s)     s - P(1)   s - P(2)         s - P(n)
```

Example:-

F(*s*)=b(s)/a(s)= (-4s+8)/(s^2+6s+8)

```
b = [-4 8];
a = [1 6 8];
[r,p,k] = residue(b,a)
r = 2×1

  -12
    8


p = 2×1

   -4
   -2


k =

    []
```

This represents the partial fraction expansion

Tf:-

SYS = tf(NUM,DEN) creates a continuous-time transfer function SYS with numerator NUM and denominator DEN. SYS is an object of type tf when NUM,DEN are numeric arrays, of type GENSS when NUM,DEN depend on tunable parameters (see REALP and GENMAT), and of type USS when NUM,DEN are uncertain (requires Robust Control Toolbox).

Example:-

```
>> n=[5 10 15];
>> d=[1 7 20 24 0];
>> sys=tf(n,d)
sys =

      5 s^2 + 10 s + 15
  ---------------------------
  s^4 + 7 s^3 + 20 s^2 + 24 s

Continuous-time transfer function.
```

tf2zp:-

[z,p,k] = tf2zp(b,a) finds the matrix of zeros z, the vector of poles p, and the associated vector of gains k from the transfer function parameters b and a. The function converts a polynomial transfer-function representation of a single-input/multi-output (SIMO) continuous-time system to a factored transfer function form.

Example:-

```
>> n=[2 8 6];

>> d=[1 6 12 24 0];

>> [z p k]=tf2zp(n,d)

z =

 -3

 -1

p =


   0.0000 + 0.0000i

  -4.5198 + 0.0000i

  -0.7401 + 2.1822i

  -0.7401 - 2.1822i

k =

2
```

Series:-

series connects two model objects in series. This function accepts any type of model. The two systems must be either both continuous or both discrete with identical sample time. Static gains are neutral and can be specified as regular matrices.

sys = series(sys1,sys2) forms the basic series connection.

Example:-

```
>> n1=[0 0 5];

>> d1=[1 1 5];

>> n2=[0 2];

>> d2=[1 2];
>> [ns,ds]=series(n1,d1,n2,d2);

>> printsys(ns,ds)

num/den =


            10

   ----------------------

   s^3 + 3 s^2 + 7 s + 10
```

parallel:-

M = parallel(M1,M2,IN1,IN2,OUT1,OUT2) connects the input/output models M1 and M2 in parallel. The inputs specified by IN1 and IN2 are connected and the outputs specified by OUT1 and OUT2 are summed.

Example:-

```
>> n1=[0 0 5];

>> d1=[1 1 5];

>> n2=[0 2];

>> d2=[1 2];

>> [ns,ds]=parallel(n1,d1,n2,d2);

>> printsys(ns,ds)

 num/den =
```

Example :-

```
 2 s^2 + 7 s + 20

  ----------------------

  s^3 + 3 s^2 + 7 s + 10
```

Feedback:-

Matlab's feedback function is used to obtain the **closed loop transfer function** of a system. Example: sys = feedback (sys1,sys2) returns a model object sys for the negative feedback interconnection of model objects sys1, sys2. To compute the closed-loop system with positive feedback, use sign = +1, for negative feedback we use -1.

Example:-

```
>> G = tf([.5 1.3],[1 1.2  1.6 0]);

T = feedback(G,1);

>> T

T =


        0.5 s + 1.3

  ---------------------------

  s^3 + 1.2 s^2 + 2.1 s + 1.3

 Continuous-time transfer function.
```

Step:-

Step response of dynamic systems.

[Y,T] = step(SYS) computes the step response Y of the dynamic system SYS.

The time vector T is expressed in the time units of SYS and the time

step and final time are chosen automatically. For multi-input systems,

independent step commands are applied to each input channel. If SYS has

NY outputs and NU inputs, Y is an array of size [LENGTH(T) NY NU] where

Y(:,:,j) contains the step response of the j-th input channel.

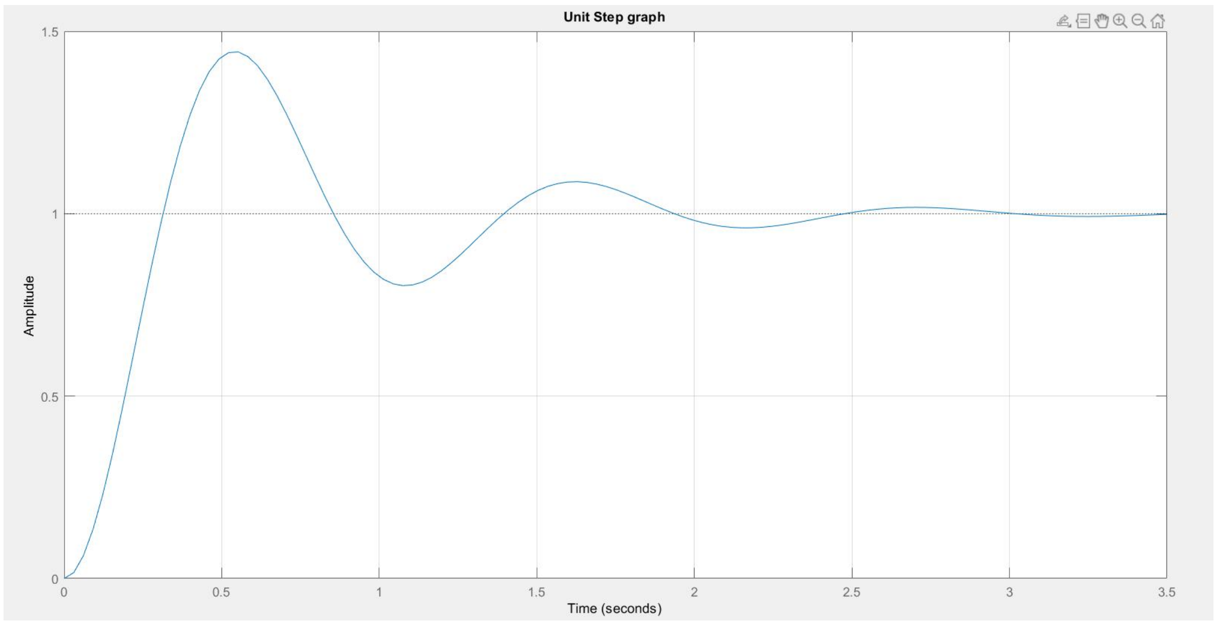
Example:-

```
>> n=[0 0 36];

>> d=[1 3 36];

>> step(n,d);

>> grid on, title('Unit Step graph')
```



Fzero:-

In Matlab, fzero functions is used **to find a point where the given objective function changes its sign.** It returns the values depending on whether the function is continuous or discontinuous in nature. Please find the below syntax that is used in Matlab: 1. a= fzero(func,a0): This is used to give a point.

Example:-

To find the value of the cos function where it is zero near to 4,

```
func = @cos; % function
a0 = 4; % initial point
a = fzero(func,a0)

>> a=4.7124
```

<u>**ANSWER NO 3**</u>

Ordinary Differential Equation (ODE) :-

The Ordinary Differential Equation (ODE) solvers in MATLAB solve initial value problems with a variety of properties. The solvers can work on stiff or nonstiff problems, problems with a mass matrix, differential algebraic equations (DAEs), or fully implicit problems. For more information, see Choose an ODE Solver.

ode45

Solve nonstiff differential equations — medium order method

Syntax

```
[t,y] = ode45(odefun,tspan,y0)

[t,y] = ode45(odefun,tspan,y0,options)

[t,y,te,ye,ie] = ode45(odefun,tspan,y0,options)

sol = ode45(___)
```

Description

[t,y] = ode45(odefun,tspan,y0), where tspan = [t0 tf], integrates the system of differential equations using Runga Kutta Method of 4rth order from range t0 to tf with initial conditions y0. Each row in the solution array y corresponds to a value returned in column vector t.

ode45 is a versatile ODE solver and is the first solver you should try for most problems. However, if the problem is stiff or requires high accuracy, then there are other ODE solvers that might be better suited to the problem.

[t,y] = ode45(odefun,tspan,y0,options) also uses the integration settings defined by options, which is an argument created using the **odeset** function. For example, use the **AbsTol** and **RelTol** options to specify absolute and relative error tolerances, or the Mass option to provide a mass matrix.

sol = ode45(___) returns a structure that you can use with **deval** to evaluate the solution at any point on the interval [t0 tf]. You can use any of the input argument combinations in previous syntaxes.

ode23

Solve nonstiff differential equations — low order method

Syntax

[t,y] = ode23(odefun,tspan,y0)

[t,y] = ode23(odefun,tspan,y0,options)

[t,y,te,ye,ie] = ode23(odefun,tspan,y0,options)

sol = ode23(___)

Description

[t,y] = ode23(odefun,tspan,y0), where tspan = [t0 tf], integrates the system of differential equations y’=f(t,y) using Runga Kutta Method of 2nd order from range t0

to tf with initial conditions y0. Each row in the solution array y corresponds to a value returned in column vector t.

[t,y] = ode23(odefun,tspan,y0,options) also uses the integration settings defined by options, which is an argument created using the odeset function. For example, use the **AbsTol** and **RelTol** options to specify absolute and relative error tolerances, or the Mass option to provide a mass matrix.

[t,y,te,ye,ie] = ode23(odefun,tspan,y0,options) additionally finds where functions of (t,y), called event functions, are zero. In the output, te is the time of the event, ye is the solution at the time of the event, and ie is the index of the triggered event.

sol = ode23(___) returns a structure that you can use with **deval** to evaluate the solution at any point on the interval [t0 tf]. You can use any of the input argument combinations in previous syntaxes.

ode113

Solve nonstiff differential equations — variable order method

Syntax

[t,y] = ode113(odefun,tspan,y0)

[t,y] = ode113(odefun,tspan,y0,options)

[t,y,te,ye,ie] = ode113(odefun,tspan,y0,options)

sol = ode113(___)

Description

[t,y] = ode113(odefun,tspan,y0), where tspan = [t0 tf], integrates the system of differential equations y'=f(t,y) from t0 to tf with initial conditions y0. Each row in the solution array y corresponds to a value returned in column vector t.

[t,y] = ode113(odefun,tspan,y0,options) also uses the integration settings defined by options, which is an argument created using the odeset function. For example, use the **AbsTol** and **RelTol** options to specify absolute and relative error tolerances, or the Mass option to provide a mass matrix.

[t,y,te,ye,ie] = ode113(odefun,tspan,y0,options) additionally finds where functions of (t,y), called event functions, are zero. In the output, te is the time of the event, ye is the solution at the time of the event, and ie is the index of the triggered event.

sol = ode113(___) returns a structure that you can use with deval to evaluate the solution at any point on the interval [t0 tf]. You can use any of the input argument combinations in previous syntaxes.

Partial Differential Equation (PDE) :-

In a Partial Differential Equation (PDE), the function being solved for depends on several variables, and the differential equation can include partial derivatives taken with respect to each of the variables. Partial differential equations are useful for modelling waves, heat flow, fluid dispersion, and other phenomena with spatial behavior that changes over time.

What Types of PDEs Can You Solve with MATLAB?

The MATLAB PDE solver pdepe solves initial-boundary value problems for systems of PDEs in one spatial variable x and time t. You can think of these as ODEs of one variable that also change with respect to time.

pdepe uses an informal classification for the 1-D equations it solves:

Equations with a time derivative are parabolic.

Equations without a time derivative are elliptic.

pdepe requires at least one parabolic equation in the system. In other words, at least one equation in the system must include a time derivative.

pdepe also solves certain 2-D and 3-D problems that reduce to 1-D problems due to angular symmetry (see the argument description for the symmetry constant m for more information).

Partial Differential Equation Toolbox extends this functionality to generalized problems in 2-D and 3-D with Dirichlet and Neumann boundary conditions.

Solving 1-D PDEs

pdepe

Solve 1-D parabolic and elliptic PDEscollapse all in page

Syntax

sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)

sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)

[sol,tsol,sole,te,ie] = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)

Description

sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan) solves a system of parabolic and elliptic PDEs with one spatial variable x and time t. At least one equation must be parabolic. The scalar m represents the symmetry of the problem (slab, cylindrical, or spherical). The equations being solved are coded in pdefun, the initial value is coded in icfun, and the boundary conditions are coded in bcfun. The ordinary differential equations (ODEs) resulting from discretization in space are integrated to obtain approximate solutions at the times specified in tspan. The pdepe function returns values of the solution on a mesh provided in xmesh.

sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options) also uses the integration settings defined by options, which is created using the odeset function. For example, use the **AbsTol** and **RelTol** options to specify absolute and relative error tolerances.

[sol,tsol,sole,te,ie] = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options) also finds where functions of (t,u(x,t)), called event functions, are zero. In the output, te is the time of the event, sole is the solution at the time of the event, and ie is the index of the triggered event. tsol is a column vector of times specified in tspan, prior to the first terminal event.

ANSWER NO 4

The point of fuzzy logic is to map an input space to an output space, and the primary mechanism for doing this is a list of if-then statements called rules. All rules are evaluated in parallel, and the order of the rules is unimportant. The rules themselves are useful because they refer to variables and the adjectives that describe those variables. Before you can build a system that interprets rules, you must define all the terms you plan on using and the adjectives that describe them. To say that the water is hot, you need to define the range within which the water temperature can be expected to vary as well as what you mean by the word hot.

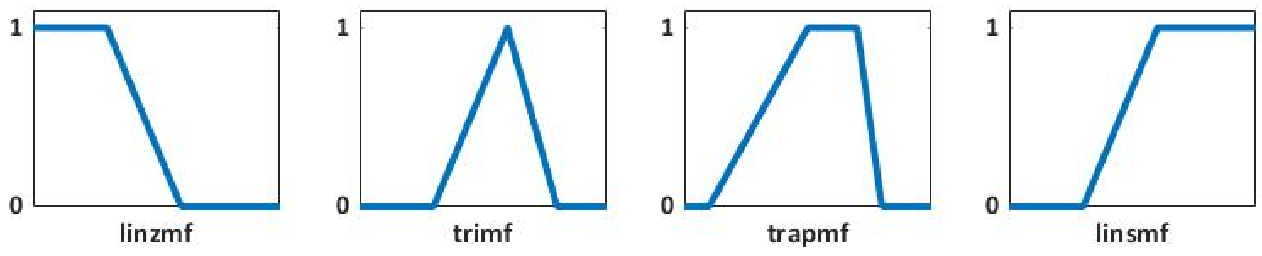
In general, fuzzy inference is a method that interprets the values in the input vector and, based on some set of rules, assigns values to the output vector.

Fuzzy Logic Toolbox software includes 13 built-in membership function types. These functions are, in turn, built from several basic functions.

- Piecewise linear functions
- Gaussian distribution function
- Sigmoid curve
- Quadratic and cubic polynomial curves

The simplest membership functions are formed using straight lines. These straight-line membership functions have the advantage of simplicity.

- trimf — Triangular membership function
- trapmf — Trapezoidal membership function
- linzmf — Linear z-shaped membership function open to the left
- linsmf — Linear s-shaped membership function open to the right



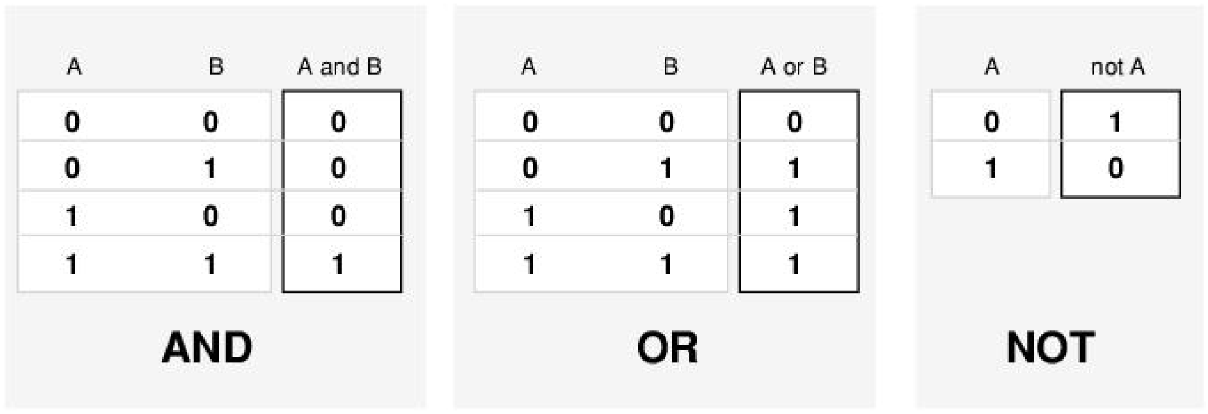
You can also create smooth membership functions using polynomial-based curves that are named for their shapes.

- zmf — Z-shaped membership function open to the left
- smf — S-shaped membership function open to the right
- pimf — Pi-shaped membership function, which is the product of an s-shaped and z-shaped membership function

Logical Operations

Now that you understand the fuzzy inference, you need to see how fuzzy inference connects with logical operations.

The most important thing to realize about fuzzy logical reasoning is the fact that it is a superset of standard Boolean logic. In other words, if you keep the fuzzy values at their extremes of 1 (completely true), and 0 (completely false), standard logical operations hold. As an example, consider the following standard truth tables.



A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

OR

A	not A
0	1
1	0

NOT

Considering that, in fuzzy logic, the truth of any statement is a matter of degree, can these truth tables be altered? The input values can be real numbers between 0 and 1. What function preserves the results of the AND truth table (for example) and also extend to all real numbers between 0 and 1?

One answer is the min operation. That is, resolve the statement A AND B, where A and B are limited to the range (0,1), by using the function min(A,B). Using the same reasoning, you can replace the OR operation with the max function, so that A OR B becomes equivalent to max(A,B). Finally, the operation NOT A becomes equivalent to the operation 1−A. The previous truth table is completely unchanged by this substitution.

A	B	min(A,B)
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	max(A,B)
0	0	0
0	1	1
1	0	1
1	1	1

OR

A	1 - A
0	1
1	0

NOT